1-1-2007

Software profiling for an FPGA-based CPU core.

Jason G. Tong
University of Windsor

Software Profiling For An FPGA-Based CPU Core

by

Jason G. Tong

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Electrical and Computer Engineering
in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2007

Your file Votre référence
ISBN: 978-0-494-34988-5
Our file Notre référence
ISBN: 978-0-494-34988-5

Abstract

Profiling tools are computer-aided design (CAD) tools that help in determining the computationally intensive portions in a software program. They are used by embedded system designers to choose computationally intensive functions of the software program for hardware implementation and acceleration. This thesis presents a detailed discussion of the various profiling tools available for embedded system design. In addition, a FPGA-BP tool, the *Airwolf* Profiler, was developed and used to profile a set of software benchmarks. The accuracy of the profiled results was compared against a well-known software-based profiling tool, GNU's *gprof*. It is shown that *Airwolf* provides up to 66.2% improvement in accuracy of profiled results and reduces the run time performance overhead, caused by software-based profiling tools, by up to 41.3%. This helps embedded designers in choosing the computationally intensive functions for hardware acceleration.

To my family for their unending love and support.

Acknowledgments

The day is finally here! I have successfully completed one of my life-time achievements, a Master's Degree in Electrical and Computer Engineering. There are several people who I would like to acknowledge in this dissertation.

First and foremost, I would like to give my sincerest thanks to my supervisor, Professor Mohammed A. S. Khalid. I am indebted for his invaluable advice, encouragement, moral support and guidance throughout my Master's research. His professionalism, knowledge and expertise will never be forgotten. I will always value our research discussions that we had over the last few years. Next, I would like to thank my thesis committee members: Professors Narayan Kar and Nader Zamani, for their invaluable suggestions, and support throughout this project. Special thanks to Professor Huapeng Wu for his valuable time chairing the M.A.Sc. Defence. Also, I would like to give a very special thank you to Lesley Shannon and Blair Fort from University of Toronto for their invaluable advice, time and assistance in this project.

My friends and colleagues from Professor Khalid's Research group (in order of appearance): Kevin Banovic, Amir Yazdanshenas, Ian Anderson, Raymond Lee, and Marwan Kanaan. I thank you all for being the greatest "cell"-mates and making my experience in EH107D and EH268 an enjoyable one. My sincere thanks go out to

Kevin Biswas, Harb Abdul-Hamid, Matthew Meloche, Ashkan Hosseinzadeh Namin, Mitra Mirhassani, Mahzad Azarmehr, Ali Bidabadi, Josh Daniel and Natalia Salgo for their friendship and support during my stay.

My heartfelt thanks go out to Lisa Price, for her editing skills and great patience in revising a majority of my papers over the years, including this thesis. Also for her continuing friendship and support she has given to me.

To Ralene Marcoccia, the Altera University Program, and the Altera Corporation, I thank you for providing the Nios II Development FPGA boards and the full licenses for the development software.

Finally and most importantly, I am indebted to my parents Yim and May Tong for their everlasting love, understanding and moral support throughout my Master's journey. This voyage would not have been easy to embark on without them.

Financial and equipment support of this research was provided by the Natural Sciences and Engineering Research Council (NSERC) of Canada, Canadian Microelectronics Corporation (CMC) and the University of Windsor.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Definition
AIB	Avalon Interface Bus
AMD	Advanced Micro Devices
API	Advanced Programming Interface
ASIC	Application Specific Integrated Circuit
CAD	Computer Aided Design
CE	Counter Enable
CPE	Computing Processor Element
CPU	Central Processing Unit
D$	Data Cache
DSP	Digital Signal Processing
DTLB	Data Translation Lookaside Buffer
FCN	Function
FLAT	Frequent Loop Analysis Tool
FLC	Frequent Loop Cache
FPGA	Field Programmable Gate Array
FPGA-BP	Field Programmable Gate Array-Based Profiling
FSL	Fast Simplex Link
HCBP	Hardware-Counter Based Profiling
HCEL	Hits Counter Enable Line
HDL	Hardware Description Language
I$	Instruction Cache
IC	Integrated Circuit
IDE	Integrated Development Environment
IP	Intellectual Property
ISR	Interrupt Service Request

ISS	Instruction Set Simulator
LSW	Least Significant Word
MSW	Most Significant Word
Nios-II-PE	Nios II Profiling Environment
PAPI	Performance Advanced Programming Interface
PBD	Platform Based Design
PC	Program Counter
PMA	Page Migration Approach
RAM	Random Access Memory
SBB	Short Backwards Branch
SBMP	Software-Based Memory Profiling
SBP	Software-Based Profiling
SOF	Static-RAM Object File
SOPC	System On Programmable Chip
SOT	Sampling Over Time
SPM	Software Profiling Methodology
TCE	Time Counter Enable
TCEL	Time Counter Enable Line
UART	Universal Asynchronous Receiver Transmitter
WOoDSTOCK	Watches Over Data STreaming On Computing element linKs

Chapter 1

Introduction

1.1 Profiling Tools for FPGA-Based Embedded Systems

In recent years, embedded systems have grown in popularity due to their increased processing power. They are prevalent in our modern society, where these systems are used in a wide variety of applications ranging from the performance of simple everyday tasks to product manufacturing. Commonly used embedded systems include cell phones, electronic pagers, television remote controls, digital cameras, personal data assistants, DVD players, HDTV and much more. In large industrial companies, embedded systems are used as programmable controllers for manufacturing, nuclear power generation, transportation and medical instrumentation.

These embedded systems consist of a hardware platform and software code working together to execute specific computation, control and communication tasks. A typical embedded system contains a processor core, memory storage and general in-

put/output interfaces. 99% of the current microprocessors produced are used for embedded systems applications [67]. The purpose of these systems is to execute software application code that is stored in memory. Due to the limitations in the hardware resources of these systems, they cannot be as flexible and reprogrammable as a desktop computer. Desktop computers are general-purpose computers containing various hardware components which can be programmed to implement any application or function. Embedded systems have dedicated and limited hardware resources that are designed specifically for performing the tasks that are specific to a particular application.

The continuing advancement and innovation of embedded systems, resulting in increased complexity, has led designers to significantly intensify their development efforts during the design process. In addition to the added difficulty, consumer demand for these devices continues to rise, which has helped to shorten design cycles and tighten time-to-mark deadlines. The design of embedded systems is becoming significantly difficult without the use of computer-aided design (CAD) tools that can effectively partition the components into the hardware or software domains. There are other added constraints that designers must consider, such as the reduction of Integrated Circuit (IC) chip area and system power consumption while sustaining maximum performance [70].

The entire objective in the development of embedded systems is to create an efficient, optimized and a balanced hardware-software partition. It involves of placing certain components in the hardware and software domains. Each of these hardware and software components execute concurrently to implement a function. The hardware-software partition determines the quality of the embedded system based on its performance. There are automated partitioning algorithms, however they require information on the system's performance prior to partitioning the embedded system's components [63]. This is where profiling tools become vital since they de-

termine which components are the performance bottlenecks and which components meet the timing requirements.

Profiling tools are CAD tools that measure the performance of a software or hardware system based on the time needed to perform certain functions. They also help in detecting problems such as communication bottlenecks in a system, cache misses and other important measurable performance metrics. They allow early detection of performance bottlenecks and help the embedded system designers to optimize their designs in order to meet system performance constraints [60, 51].

There are several profiling tools available today that can be used to profile software code running on a target processor. These tools provide different profiling information that can benefit embedded designers so that they can optimize the software code. Despite the variety of profiling tools that are available, many of them use different measuring techniques that can potentially provide inaccurate feedback. The majority of the profiling tools used are software-based, which require the designer to compile their software programs to include instrumentation code at the binary level. This is not desirable since it is very intrusive to the original program and can cause unpredictable execution behaviour of the software. Sampling techniques are also used in a variety of profiling tools and can provide varying results depending on the sampling frequency of the profiler. This consequently affects the accuracy of the profiled results, which can potentially lead embedded designers to implement the wrong software functions in hardware. It is imperative that profiling tools minimally disturb the original program binary file and have the ability to provide accurate results in order to create an effective hardware-software partition of the embedded system.

1.2 Thesis Objectives

The work presented in this thesis conforms to the following objectives:

1. To create a minimally intrusive profiler that does not require the insertion of instrumentation code added to a software program's binary file. This profiler should be able to accurately measure the amount of time a software function has taken to execute on a target processor.

2. Use the developed profiler to profile several common software benchmarks running on an FPGA-based soft-core processor system.

To satisfy the first objective, an Field Programmable Gate Array (FPGA)-based on-chip profiler, called the *Airwolf* profiler, was developed. This profiler contains twenty profiling counters that can measure the performance of up to twenty different software functions. It is minimally intrusive and collects profiling information by measuring the number of system clock ticks that each software function takes to execute on a soft-core processor. For the second objective, a profiling environment was developed that is based on the Altera Nios II soft-core processor [32]. This environment was used to execute several software benchmarks and to profile them using the *Airwolf* profiler. The results obtained using the *Airwolf* profiler were compared against those obtained from the GNU's *gprof* [36] software-based profiler. The results collected using the *Airwolf* profiler show a significant increase in profiling accuracy over those of the *gprof* profiler.

This entire project emphasizes the use of FPGAs in the design of embedded systems. FPGAs have grown in size in terms of logic capacity and on-chip memory resources. This enables them to implement and rapidly-prototype large digital circuits such as those commonly encountered in embedded systems design without the need of fabricating the system onto an Application Specific Integrated Circuit (ASIC). The supporting CAD tools enable designers to quickly create embedded systems by instantiating a set of Intellectual Property (IP) components and automatically connecting them to the peripheral components and programing the FPGA board.

1.3 Thesis Organization

This thesis contains six chapters. Chapter 2 covers the various design methodologies for embedded system design. Chapter 3 presents a survey of the profiling tools that are available. Chapter 4 introduces the *Airwolf* Profiler and discusses its architecture and components. Chapter 5 presents the experimental framework used to obtained profiling results and presents a discussion on these results. Chapter 6 provides concluding remarks and a discussion of future work.

Chapter 2

Design Methodologies for Embedded Systems

The development of embedded systems involves the combination of hardware and software components together to meet the requirements of a specific application. There are several design methodologies that can help embedded designers to coordinate different design tasks in order to meet tight time-to-market deadlines and to fulfill all the specified performance requirements. These are:

- Traditional Design Methodology
- Hardware-Software Co-Design
- Functional Architecture Co-Design
- Platform-Based Design

In this chapter a brief introduction to these methodologies is provided so that the reader is able to understand the different approaches that are used in the design of embedded systems.

2.1 Traditional Design Methodology

The Traditional Design Methodology [39] is a set of design approaches that are commonly used in the automotive industry [54]. This approach usually follows a waterfall model of system development [69].

Figure 2.1 shows a flowchart for the traditional methodology for the design of embedded systems. Initially a set of specifications are defined which describe the system's operations and the performance requirements that the system must satisfy. After this initial step, the hardware and software components are designed independently. Usually a group of hardware and software engineers develop these components distant from each other and at different times during the design process. There is very minimal interaction between these groups as the hardware architecture is being built and the software code is written. It is usually presumed that these components can be combined together without any incompatibility issues. As the components are fully synthesized and functional, the systems' components are integrated together, during what is known as the system integration stage. Following this stage is the verification and prototyping stage, during which designers verify and test the prototype. Lastly, the design is sent for fabrication.

This design methodology is suitable for smaller and simpler designs, but is not feasible for complex embedded systems. It introduces many problems and causes compatibility conflicts to occur between the software and hardware domains. When designing the hardware (or software) components first, it may be difficult to determine if the software components are able to run on the hardware architecture and vice versa. In many cases, certain hardware components may need to be changed if the software

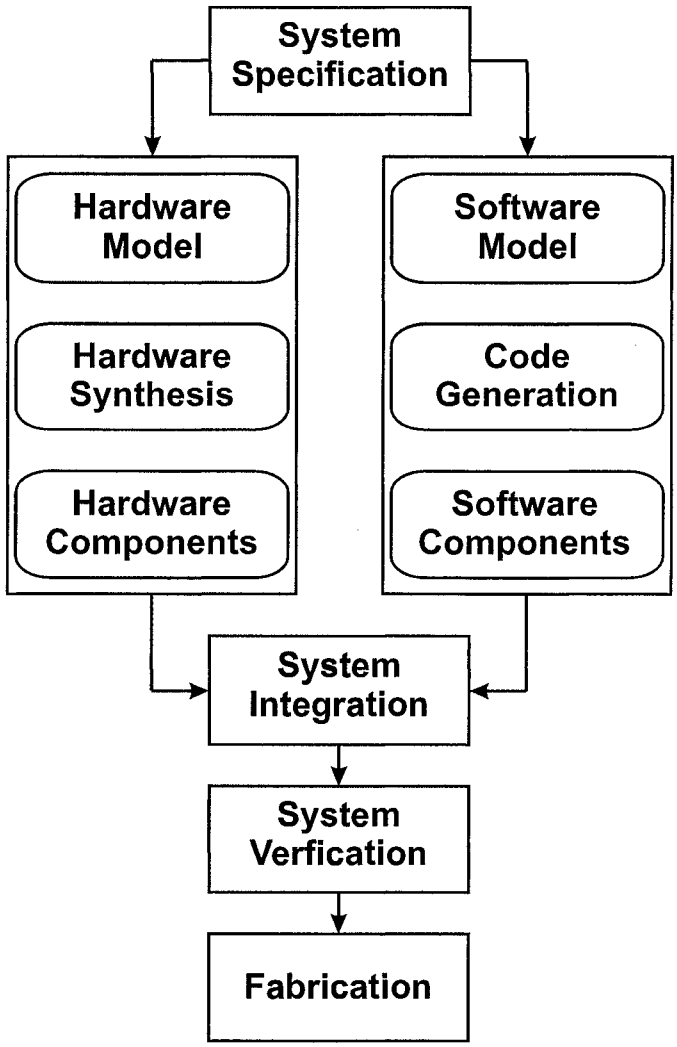

Figure 2.1: The Traditional Design Methodology.

components, which were built in a different design time-frame, rely on an unsupported hardware function (or architecture) in order to execute properly. Using the traditional design methodology, designers use most of their time on interface debugging tasks and have less time for other important tasks such as overall system verification, testing and optimization. In some cases, many design iterations may be required to meet design goals and constraints. This may lead to missed time-to-market deadlines and design obsolescence.

2.2 Hardware-Software Co-Design Methodology

The Co-Design methodology for embedded systems enables the hardware and software components to be designed concurrently. It allows designers to find an efficient and balanced hardware-software partition of the components of the embedded system, while maintaining compatibility. This methodology ensures the hardware platform is able to execute the software components (or supporting application software) and has the necessary computing resources for proper execution.

One of the main advantages of the co-design methodology is the ability to detect early compatibility issues in the design. When problems are detected earlier in the design stage, they are easier and less expensive to fix [55].

There are many proposed co-design methodologies and the majority of them have focused on the implementation of digital signal processing algorithms or embedded systems design [25]. In each of the methodologies, most have common design stages that will eventually lead to a system that performs a specific function or application. A flowchart for the hardware-software co-design methodology is shown in Figure 2.2 [30].

The co-design process starts with the specification of the system, usually expressed using a high level system modeling language or a software program. This defines the requirements, design constraints and the functionality of the system. Next, the

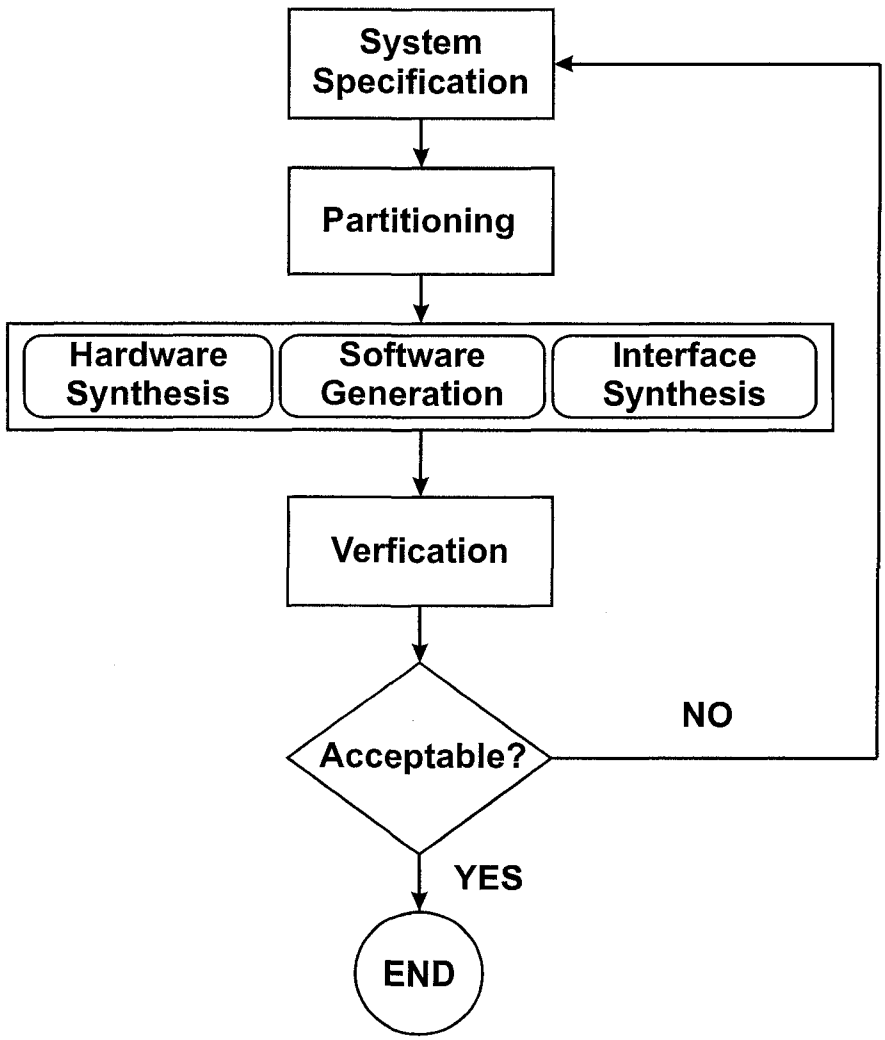

Figure 2.2: The Hardware-Software Co-Design Methodology

hardware-software partitioning stage determines which functions or components are to be placed in the hardware domain and which are handled by software. The third and most important stage is synthesis, in which the hardware, software and interface components are synthesized concurrently. Hardware and software engineers continuously interact with each other by exchanging performance information and functional requirements of all the components. This ensures that the hardware architecture and the software program can execute together without difficulty. Finally, the verification stage determines if the designed system meets the design requirements and performance constraints. If the design fails to meet the requirements, iteration is needed, which leads back to the review of the specifications. The number of iterations depends on the design size and complexity. The hardware-software co-design process helps minimize the number of iterations and the design time required to implement a complete system.

2.3 Function-Architecture Co-Design

Another methodology used in the design of embedded systems is the Function Architecture Co-Design [54]. In this approach the embedded system is built at a higher abstraction level, which allows designers to focus on the design of the system's functionality without having to be concerned with how that functionality is implemented. The hardware-software co-design puts emphasis on interfacing the hardware and software components together. This process, however, does not focus on the design tasks at the system-level, which often leads to extended time in reaching the target design.

Figure 2.3 illustrates the Function Architecture Co-Design [27]. The methodology starts at the specification stage where the architectural and functional descriptions of the system are defined. During the specification stage, the system is described using two different definitions: [57]:

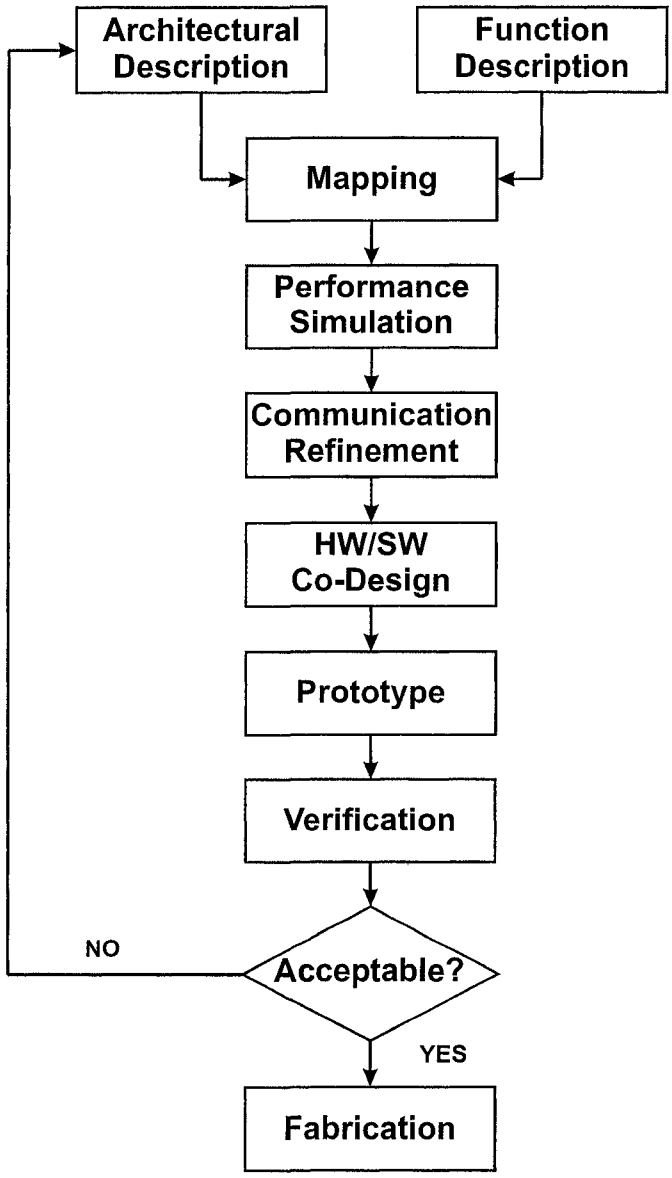

Figure 2.3: The Function-Architecture Co-Design Methodology

- Functional Definition: the specific function or application that the system will provide
- Architecture Definition: a candidate architecture that contains all the IP cores, hardware and software components that implement the specified function.

Following the specification stage is the mapping stage, in which the system's functions are partitioned and directly mapped to the chosen system architecture. In addition, the hardware and software interfaces are also mapped onto the architecture's resources. The performance simulation stage is next, which involves carrying out all of the simulations for each component, and performing various verification techniques on the mapped hardware and software components. This is done to verify that the mapped system is functional and is capable of meeting the design constraints. The next stage is the communication refinement stage, in which the inter-communication between the various system functions are defined [57]. Once these modelling stages are completed, the system design goes into a hardware-software co-design synthesis where the components of the system are synthesized together. At this stage, the prototype of the embedded system has been constructed, and then goes into the verification stage. Further design iterations are performed if the system does not meet the specified design requirements. Fabrication is the last stage, in which the verified system is taken and sent off for production.

2.4 Platform-Based Design

The Platform-Based Design (PBD) methodology emphasizes the use of reusable IP cores as a platform upon which designs are constructed [54]. This involves a design-space exploration that attempts to find a balance between a hardware platform consisting of a set of instantiated programmable IP cores and the ability of the architecture to support a set of applications. Platform-Based Design uses a "meet in the

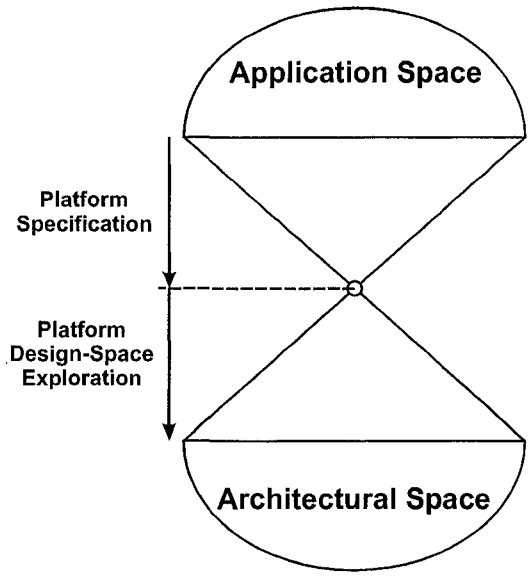

Figure 2.4: Design Space Exploration

middle approach" [26] as shown in Figure 2.4 [56].

There are two different approaches used in PBD: the top-down approach and bottom-up approach. Using the top-down approach, the system's platform architecture, including the processor's speed, memory capacity and other instantiated peripherals, are defined at the beginning of the design cycle. The bottom-up approach defines a family of different software applications that can be programmed on the given hardware platform. The intersection of the architecture space and the application space defines the hardware platforms available for a set of applications. In some cases, the hardware platform that was derived may be over-designed for the particular application, although this is deemed beneficial to designers since they can create new software products and extend the useful life of the hardware platform [49]. This implies that using platform-based design for embedded systems emphasizes the reuse of existing components. Not only does this reduce the amount of hardware resources used but can help to minimize the cost of manufacturing the embedded system.

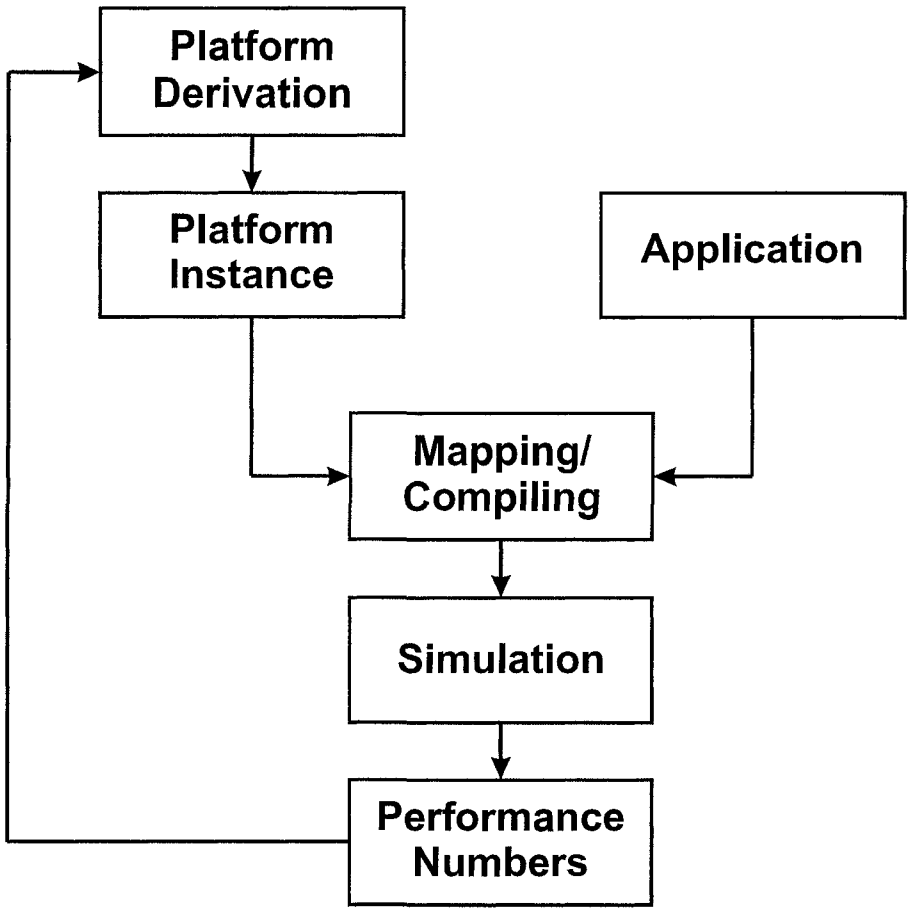

Figure 2.5: Platform Based Design

Figure 2.5 describes the Platform-Based Design methodology of embedded systems [54]. The designer starts by specifying the platform architecture, which outlines the performance constraints and the functionality of the entire system based on the intended application. This includes the specification of the required speed of the microprocessor, memory capacity, cache memories, etc. From the defined requirements, a platform instance is made which contains all of the instantiated hardware components and software programs required to execute a specific application. Following this stage is the mapping and compiling of the system, which includes hardware platform synthesis and the program code generation. Next, the compiled system goes into the simulation stage, when designers test all of the components to ensure that they are functioning correctly and meeting the design constraints. Based on the performance numbers retrieved from the simulation stage, the designer can determine if the system has satisfied the specified requirements. If not, the system goes into another design iteration cycle until it has fully met all of the constraints.

2.5 Summary

This chapter presented an introduction to different embedded system design methodologies. In the next chapter, a discussion of profiling tools is presented.

Chapter 3

Profiling Tools

There is a wide variety of profiling tools available that measure different performance metrics and retrieve diverse sets of profiling information. Section 3.1 discusses profiling tools and a proposed software profiling methodology for the design of embedded systems. The subsequent sub-sections classify the different types of profilers available as follows: *Software-Based Profiling* (SBP) Tools, *Software-Based Memory Profiling* (SBMP) Tools, *Hardware-Counter Based Profiling* (HCBP) Tools and *FPGA-Based Profiling* (FPGA-BP) Tools. In each of these categories, a brief survey of these existing tools is presented.

3.1 Profiling Tools and the Software Profiling Methodology

There are several methodologies and approaches used in the design of embedded systems. As explained in chapter 2, the majority of the methodologies begin at the

specification stage in which all the functionalities of the system and the supporting architecture to implement that function are defined. Usually embedded designers have two options for the initial implementation of their design based on the specifications. For the first option, the embedded system can be entirely implemented in hardware while moving certain components to the software domain, depending on the execution performance of those functions [42]. The second option is to have the entire embedded system implemented in software [35] and invoke a profiler that measures the performance of the software program. The information provided by the profiler is used by designers to help them choose which software functions are more desirable for hardware implementation.

Profiling tools are used to measure the performance of a program that is running on the target processor of an embedded hardware platform. These tools provide useful information for designers so that they can identify certain software hot-spots that are causing a performance bottleneck. Designers can choose either to optimize the software code to alleviate the performance issue or implement the computationally intensive function in the hardware domain in order to achieve a speed-up in performance of the entire system. It is imperative that profilers provide accurate results and properly detect these hot-spots. This can lead to the creation of a balanced partition between the hardware and software components. The quality of the embedded system is entirely dependent on the efficiency and the effectiveness of the hardware-software partition of the system's components. The application of profiling tools has led to a proposed *Software Profiling Methodology* (SPM) as shown in Figure 3.1 [60].

The design flow is similar to the hardware-software co-design methodology of embedded systems [30], as explained in Section 2.2. The SPM begins at the software specification stage. The complete embedded system is written in a high level language such as C or C++ and then the software is functionally verified. Next a profiler is invoked in order to measure the runtime performance of the program and eventually

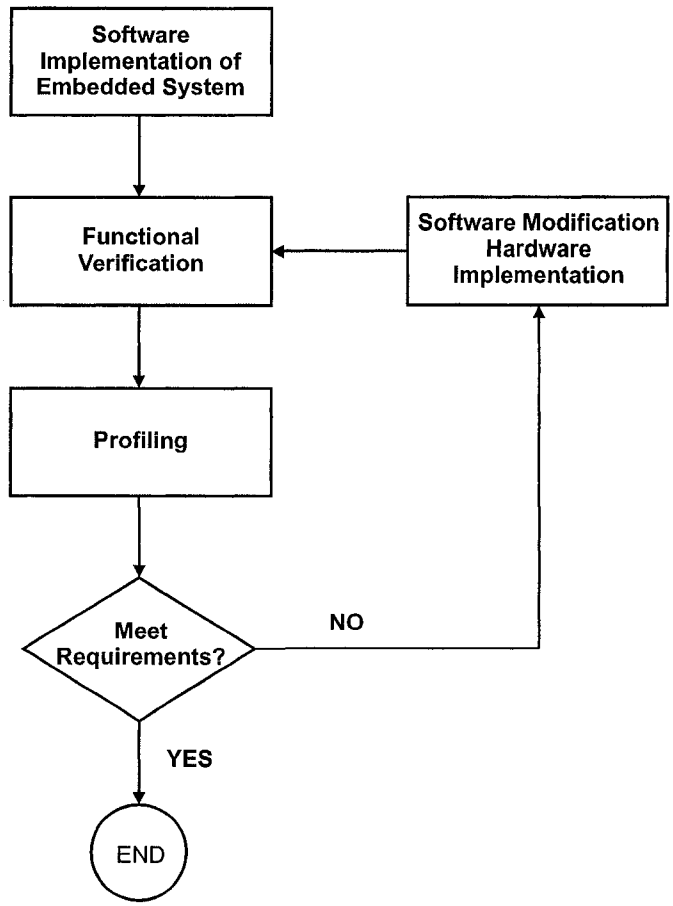

Figure 3.1: Software Profiling Methodology

return feedback and performance statistics to the designer. The designer analyzes the results and determines if the software code meets the specified performance constraints. That same profiling information can be used by an automated hardware-software partitioning CAD tool [63]. If the system fails to meet the requirements, the designer will try to optimize the code or move certain computationally intensive functions into the hardware domain as a hardware accelerator. If necessary, the entire methodology starts again until the designer is satisfied with the performance.

Existing profiling tools offer different types of profiling capabilities and support different programming languages. C/C++ profiling tools are common, but there are also tools available that can profile programs written in Java [38, 37]. Mentor Seamless Co-verification environment provides a profiler that takes a design written in SystemC [13] and measures its performance based on processor utilization, cache efficiency, memory hotspots, bus utilization and bus master contention [12].

Currently, there are many different kinds of profiling tools that are used to retrieve a variety of profiled information about a program. The most common is function-level profiling which measures the amount of time needed for a function to execute on the processor. Another type is memory-level profiling that determines which function, data variable type or instruction is causing memory related problems: excessive memory references, cache misses, heavy pointer dereferencing, branching and looping instructions. Figure 3.2 depicts the proposed classification of profiling tools. There are three main categories: *software-based*, *hardware-based* and *FPGA-based*. We describe each of these in detail in the following sections.

3.2 Software Based Profiling (SBP) Tools

Software-Based Profiling (SBP) is the most common technique for measuring the performance of application code written in a programming language. There are two approaches to profiling the software code when using these tools: simulation and the

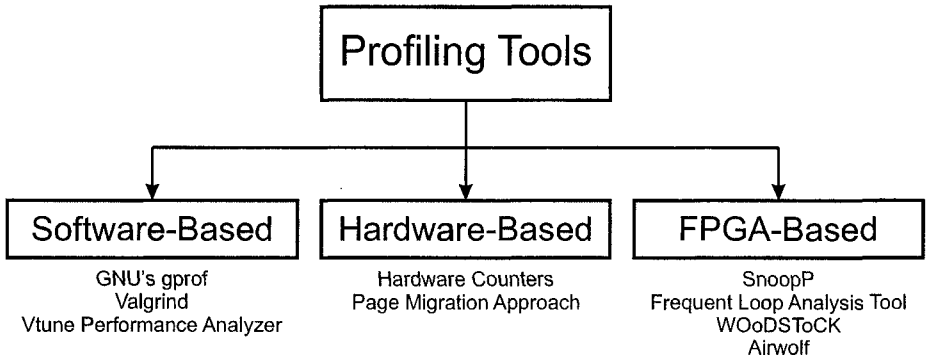

Figure 3.2: Profiling Tool Classification

insertion of instrumentation code. Simulations take place in virtual environments that simulate the behaviour of a microprocessor as the software code is running on a virtual environment. The insertion of instrumentation code allows an SBP tool to attach itself to the binary file and collect performance information during the execution of a program on the processor. In this section, we describe an ISS, GNU's *gprof* [36] and Intel's [11] *Vtune* [45] is given.

3.2.1 Instruction Set Simulator

Instruction Set Simulators (ISS) are one of the SBP tools used for profiling software code running in a simulated environment. One popular ISS is the *SimpleScalar* Toolset which simulates application code running on the *SimpleScalar* computer architecture [29]. The advantages of using an ISS for profiling is that the designer is able to view the entire data flow movement inside the microprocessor's registers during the simulation. It keeps track of all of the execution processes, the current instruction in execution, data manipulations, cache accesses and other reportable events. This does not require the software code to be modified, therefore intrusiveness to the binary file is non-existent.

The use of an ISS may not be feasible for larger software programs or with system-

on-a-chip designs since they can be very slow to simulate [51]. This could lead to very inaccurate profiles of the execution times of each function. Simulations can have varying times to complete depending on the complexity of the software code. It may take several hours to run an entire simulation which may only cover a few seconds of real-time, thus misrepresenting the entire execution time. Due to the increasing complexity of embedded systems designs, constructing complex models of the system's components and other external environments may not be possible

3.2.2 GNU's gprof

gprof [36] is an open-source profiling tool that is used on Linux [5] and Unix [6] workstations to profile C and C++ application code. It provides two types of profiled outputs: the flat profile and the call graph. The flat profile is a report of how much time the program is spent on each function and the number of times that function was called. The call graph displays each function, its calling function and other functions called within that function. To utilize this profiler, the designer is required to compile the code with the default debug instrumentation setting. This option inserts additional instrumentation code into the binary executable file, as required by *gprof*.

During program execution, *gprof* utilizes the inserted instrumentation code to monitor the performance of the program running on the Central Processing Unit (CPU). The instrumentation code allows *gprof* to count the precise number of function calls and generate the appropriate number of interrupts to sample the program counter (PC) of the CPU. It is capable of generating a profile that accurately counts the number of functions that have been called, however, the reported execution time of each function may be somewhat inaccurate.

gprof collects information on the execution time of a program by reading the value of the PC at specified intervals. The PC value determines which function is being

executed on the processor. Based on this value, *gprof* increments the execution time counter of the function that is currently executing by its sampling period. This can create inaccurate timing results for each function called and the execution time of the entire program [68]. The accuracy of the profiled execution time is entirely dependent on the sampling frequency of the PC.

3.2.3 Intel's VTune

Intel's *VTune Performance Analyzer* is an SPB tool that profiles C/C++ code that is executed on Intel processors [45, 47, 11]. The *VTune* analyzer features three profiling modes: *Sampling Over Time* (SOT), *Call Graph* and *Counter Monitor*. Each of these modes is discussed briefly in the following paragraphs.

There are two sampling methods that are used by *VTune*: *Sampling Over Time* (SOT) and the *Pause/Resume Application Programming Interface* (API) [24]. SOT profiles the software code and shows the performance results specified "over time" of each thread, function and instruction until the program has completed execution. In addition, it can detect when the processor is in an idle state. This allows designers to optimize the application code to execute other threads when the processor is not executing any threads.

Sampling using the *Pause/Resume API* [24] requires the user to insert certain functions into various parts of the software code. Such functions are `VTPause()`, `VTResume()`, `VTPauseSampling()`, `VTResumeSampling()`, `CMPause()` and `CMResume()`. These functions are used to select certain code regions for profiling.

VTune's Call Graph profiler [58] displays the calling sequences of functions during execution of the software code. The *VTune* profiler adds instrumentation code into the binary executable file so that it can monitor and identify the number of specific functions called during run-time. Additionally, it identifies the critical path in the call graph which displays the potential bottlenecks that limit system performance.

3.2.4 Summary of SBP Tools

The use of the sampling technique in common software-based profilers helps to reduce the run-time overhead during profiling. Nevertheless, this can produce inaccurate profiled results which can potentially create a sub-optimal partition of the embedded system. The use of an ISS can also produce inaccurate results since simulators are only as good as the system model that is being simulated. Also, the simulation time may not accurately match the actual run-time execution of the program. Certain SBP tools require the designer to link their program with instrumentation code which is inserted at the binary level. This can lead to an excessive number of interrupt calls which may cause unpredictable behaviour of the software code running on the embedded hardware platform. Additionally, the instrumentation code can lead to an increase in code size and may potentially change the behaviour and the performance of the software system.

3.3 Software Based Memory Profilers (SBMP)

Embedded system software must take great care in ensuring that the memory system is used properly [33, 34]. One of the main problems is memory leakage. A memory leak is caused when the application code consumes unnecessary memory resources and fails to release the memory that is no longer in use. Prolonged memory leaks in an application can cause the system to behave unpredictably and eventually run out of memory, leading to the failure of the embedded system. An increase in the number of unnecessary memory accesses and paging is another problem, since it introduces latency in retrieving data and operands for instructions to execute on the processor. Excessive numbers of read and write accesses to memory are the most common overhead operations in CPUs [41]. These operations generally cause performance degradation. Cache misses are also an issue when the processor is unable

to retrieve instructions from its own cache memory. This is due to mispredicted branching instructions, heavily nested dereferencing of memory pointers and looping instructions.

Memory profilers are needed to detect the problems listed above, so that they can be resolved by the designer. They provide detailed information about which function call in the application code is producing memory leaks, cache misses and high memory referencing. Reducing the number of memory accesses can improve performance and minimize performance overhead [50]. In this section, the following memory profiling tools are described: *Valgrind* [14], and *Purify* [44].

3.3.1 Valgrind

Valgrind is an open-source GNU profiling tool for Linux systems [14]. This profiler can check the calls for read and writes to memory, as well as for allocating and freeing memory using functions such as the C++ functions `new` and `delete`. The major advantage of *Valgrind* is its capability for cache memory profiling. It simulates the CPU's Level 1 data and instruction level caches as well as Level 2 cache. *Valgrind* determines a cache hit count for every line of the program that is being traced and analyzed. It can profile applications of various sizes, from small functions to complex application systems.

The technique *Valgrind* uses to measure the performance of software code is to run the application in a simulated virtual processor environment. Other components and libraries of the software code are linked to the simulator as well. During the simulation process, the profiling data is collected and it is stored in a log file. The usefulness of this method depends on how well the functions and data structures are modelled in the simulator. *Valgrind* is capable of profiling memory activity on larger programs, although the performance of the software program can degrade.

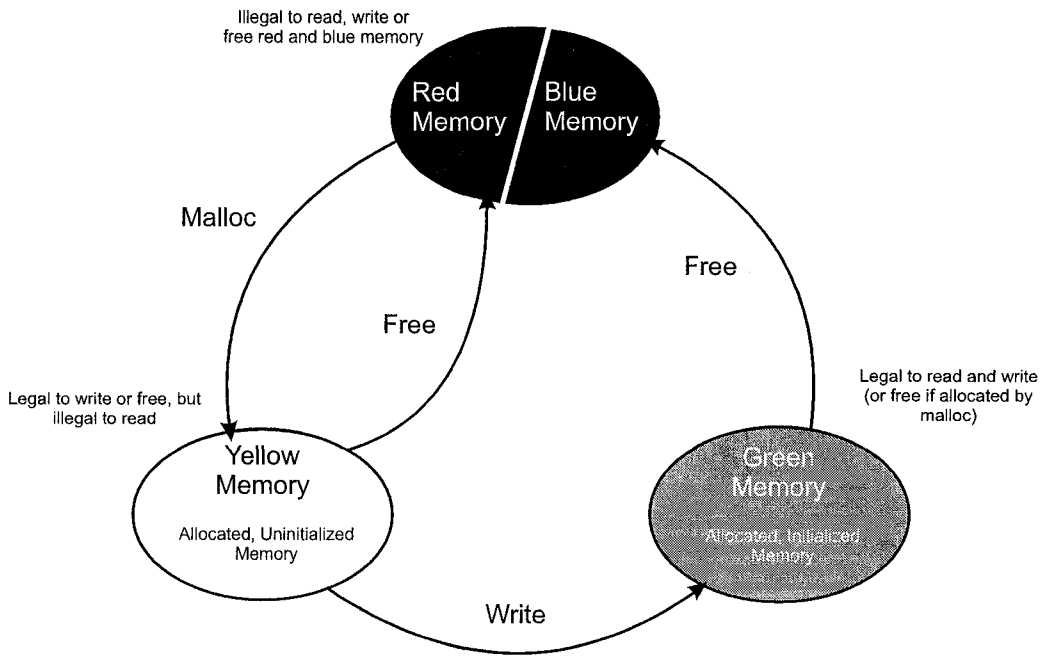

Figure 3.3: Rational Purify's Memory Profiling Colour Code

3.3.2 Rational Software's Purify

Rational Software's Purify [44] is a software-based memory profiler that can be used on Microsoft Windows [7], Unix [6] and Linux [5] operating environments. The tool helps in solving memory problems and determines the exact code location that is causing the error. The kinds of problems the program detects are memory leaks, reading and writing beyond the bounds of an array in memory, attempts to free un-allocated memory and using un-initialized memory. *Purify* uses a four colour scheme to represent memory problems as shown in Figure 3.3 [44]: red, yellow, green and blue.

The red zone indicates the program has no memory access unless memory is explicitly allocated by using a `malloc` or `new` function. *Purify* initializes all heap and stack memory as a red zone until it is allocated. The yellow zone is the memory that

is allocated by the program. It is not legal to read from it because it is not initialized or does not contain any valid data. The green zone is memory that has been written into and is available for reading and writing data. Blue zone is memory that is freed by the program and is no longer accessible.

3.3.3 Summary of SBMP Tools

Memory profiling tools are essential for detecting memory leaks, allocation and de-allocation errors, as well as instructions that cause cache read/write misses. They give the designer more options to analyze and optimize the software code prior to porting it to the target architecture. In addition, they provide more detailed performance information than function-level profilers. The problem with the current memory profiling tools is that they use the same measuring techniques as SBP tools. Some memory profilers require that the designer include instrumentation code in their application at the binary file. This introduces the issue of large code sizes and runtime overhead. Some memory profilers use sampling techniques to sample the hardware counters and retrieve their values. As discussed in the case of software-based profiling, sampling techniques can produce inaccurate results and may potentially mislead the designer to improperly implement certain functions in the hardware or software domains.

3.4 Hardware-Counter Based Profiling (HCBP) Tools

Hardware-Counter Based Profiling (HCBP) tools utilize on-chip hardware counters that are available on advanced processors such as *Sun Ultrasparc* [64], *Intel Pentium Processors* [46] and *Advanced Micro Device* (AMD) Processors [9]. These hardware counters are dedicated to monitoring specific events that occur during runtime execution of an application. The types of events which can be monitored are: memory

accesses, cache misses, pipeline stalls, types of instructions executed and etc. HCBP tools do not require the use of instrumentation code since these counters are designed to collect performance information of the software program. In addition, very little performance overhead is introduced during runtime execution.

Accessing these counters requires a unique instruction. The *Performance Advanced Programming Interface* (PAPI) [28] provides users with a high level interface to access these counters and can supports many different processors [62]. *Intel's VTune* counter monitor provides an interface for accessing and utilizing the hardware counters to profile application code executing on Pentium-based processors [46].

3.4.1 Hardware Counters Approach

Itzkowitz et al from *Sun Microsystems* have described a software profiling tool that utilizes the hardware counters in an *Ultrasparc-III* microprocessor [48]. Originally this profiling tool was built as an extension of the Sun One Studio [4] compilers and performance tools, which are used for measuring the performance of software code. These hardware counters are included in the architecture and contain different types of event counters such as, *Instructions Completed*, *Instruction-cache* (I$) *Misses*, *Data-cache* (D$) *Read Misses*, *Data-translation-lookaside-buffer* (DTLB) *Misses*, *External-cache* (E$) *References*, E$ *Read Misses*, E$ *Stall Cycles*, and many others.

There are some limitations to using this tool. One such limitation is counter-skidding. The tool uses hardware-counter overflows to obtain profiled information. When a counter overflow occurs, the tool does not execute a precisely timed trapping mechanism to obtain the correct value of the counter. The second problem is the backtracking mechanism of instructions which was implemented as a solution to solve the trapping mechanism flaw. The backtracking technique is used to find the instruction address that caused the overflow event to occur, however the instruction immediately preceding the current one in the processor's PC may not have the cor-

rect address value, due to the possibility that the previous instruction was a branch call. Instead of relying on the value of the PC, the profiling tool tries to find the proper values in other registers to calculate the effective address of the instruction that caused the overflow event. It is not guaranteed success in finding the address since the value of the registers may have changed once other overflow signals have been delivered to other hardware counters. Despite with these drawbacks, the tool has managed to find the proper instruction 99% of the time. The MCF benchmark was profiled and the feedback provided enabled a 20% performance improvement.

3.4.2 Page Migration Approach

The *Page Migration Approach* (PMA), developed by *Tikir et al* utilizes hardware-counters for profiling memory with memory page-migrating capabilities [65]. The profiler was used on a multi-processor system based on *Sun's SunFire Server* as shown in Figure 3.4. Each system board contained several processors and memory. The *Sun Fire Link* hardware counters are used to sample the frequency with which each processor "touches" a page of memory that is remote from the on-board local memory hardware. At a certain number of counts specified by the user for remote touching of memory pages, the profiler halts the execution. It then migrates that particular memory page to the processor that accesses it most frequently for read and write operations. PMA has demonstrated 90% speed improvement when certain memory pages are placed closest to the processor that requires data from that page.

3.4.3 Desktop Processor Profiling Counters

There are consumer desktop processors today that contain hardware counters which monitor the performance of application code in the CPU. *AMD Athlon* microprocessors [9] contain four 48-bit performance hardware counters that can be used as event driven or timing driven counters. These counters can monitor the number of times a

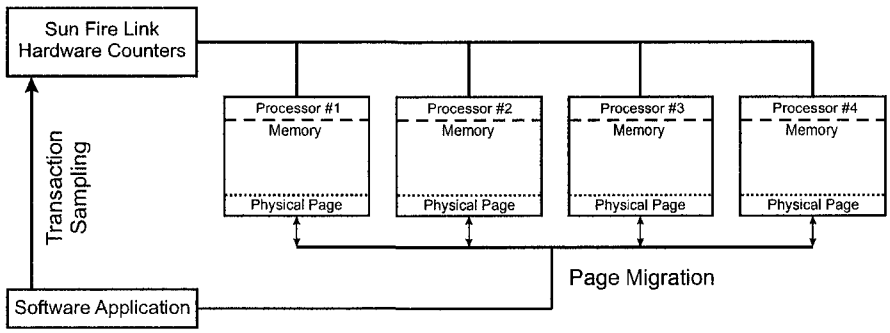

Figure 3.4: Page Migration Approach

certain event occurs or they can measure the duration of an event that is currently taking place on the processor. *Intel Pentium* microprocessors also contain a set of performance hardware counters [46]. They are also event or timing driven and are accessible through *Intel's VTune* [45] profiling tool.

3.4.4 Summary of HCBP Tools

Using hardware counters for profiling software code is beneficial since it does not introduce any instrumentation code, leaving the compiled application source code untouched. Additionally, they do not add any performance overhead since the data collection of these counters occurs during runtime execution of the software. However, there are drawbacks when using HCBP tools. First, some HCBP tools may require the user to reconfigure and reprogram the counters to detect different events, which can lead to the addition of certain functions at the source code level. Secondly, they use the sampling method to sample the hardware counters which leads back to the problems that were introduced with SBP tools. Thirdly, handling of interrupts affect the gathered data since the interrupt service routines (ISR) used add to the number of events. Lastly, there is a limited number of hardware counters available. The programmer must run the application many times to obtain data for different

monitoring events [62].

3.5 FPGA-Based Profiling (FPGA-BP) Tools

FPGAs are user programmable integrated circuits that offer reasonably high level of integration, negligible prototyping cost and instantaneous manufacturing capability. Riding on Moore's law [52], FPGAs have grown in logic capacity while maintaining an affordable cost for many applications [31]. Embedded development kits that utilize FPGAs contain an abundance of on-board resources such as clock multipliers, fast memory chips, math co-processors, etc. This makes them an attractive alternative for rapid prototyping of large embedded system designs due to their reconfigurability and flexibility that they offer to the designer.

Researchers today are developing profiling tools that can help designers working on embedded system designs using FPGAs. The two major FPGA vendors, Altera Corporation [17] and Xilinx Incorporated [72], provide embedded system development kits which use the Nios II [32] and MicroBlaze [73] soft-core processors, respectively. These soft-core processors are instantiated on the FPGA and used as basic building blocks for designing embedded systems [66].

FPGA-based profiling (FPGA-BP) tools also utilize these soft-core processors for profiling. In FPGA-BP tools, the designer executes the software on the soft-core processor and collects the performance data provided by the on-chip profiling hardware. These tools have provided improved results compared to the previous profiling tools described earlier. They keep latency and performance overhead at a minimum, because they are non-intrusive and require negligible instrumentation. They do not use the sampling technique and require very minimal processor computation. These features are highly desirable for profiling tools used in embedded systems. In this section, a detailed discussion of the existing FPGA-based profiling tools is provided.

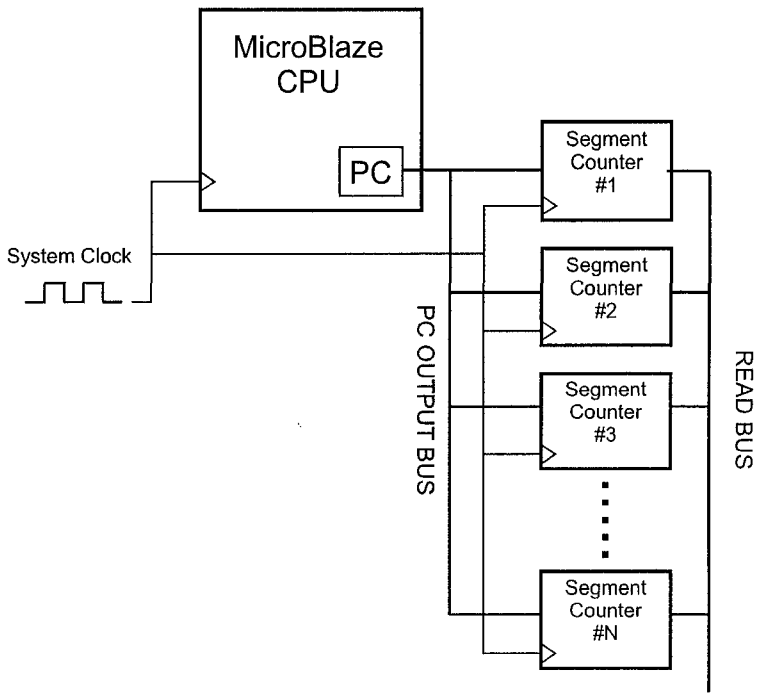

Figure 3.5: Snoopy's Profiling Architecture

3.5.1 SnoopP

SnoopP [60] is an on-chip function-level profiler that was implemented on the Xilinx Virtex-II 2000 FPGA board. This board is used to implement designs based on Xilinx MicroBlaze [73] soft processor. The on-chip profiler utilizes the MicroBlaze as a target processor. *SnoopP* uses a hardware profiling architecture that is non-intrusive to the code, such that any additional instructions, commands or other flags are not necessary. Figure 3.5 depicts the hardware architecture for the *SnoopP* profiler.

SnoopP consists of a variable number of segment counters that are user specified and define the address of instructions to be analyzed. The number of segment counters is dependent on the number of functions the user wishes to profile and the area available on the FPGA.

The segment counters, shown in Figure 3.6, determine if the value of the PC

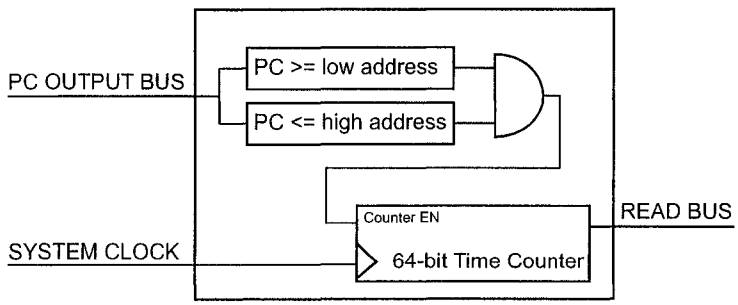

Figure 3.6: Snoopy's Profiling Counter

address is in the range of memory addresses in which the binary code corresponding to the function resides. This is determined by the comparators inside each segment counter. If this condition is true, the comparator sends an enable signal to the hardware counter which utilizes the processor's system clock to count the number of clock cycles the function has used. This gives the designer the precise number of clock cycles that the particular function needs to execute on the processor. *SnoopP's* and *gprof's* results were compared, and it was shown that *SnoopP* was significantly more accurate. Additionally, *SnoopP* does not slow down the performance of either the software or the profiling process.

3.5.2 Frequent Loop Analysis Tool (FLAT)

Frequent Loop Analysis Tool (FLAT) is a tool that detects functions in software that heavily use loops [40]. In most cases, loops use 90% of the execution time while constituting only 10% of the entire software code. FLAT searches for these critical regions and records the execution frequency of each loop-intensive function into a cache-like hardware architecture that is implemented in an FPGA. A block diagram of the FLAT architecture is shown in Figure 3.7.

Usually a loop instruction is typically denoted as a *Short Backwards Branch* (SBB), when the program jumps back to the first instruction of that loop. The

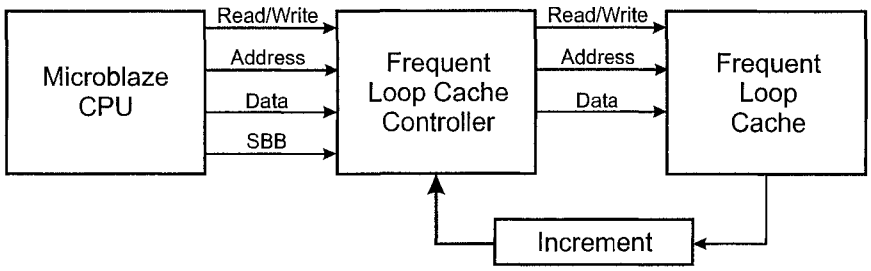

Figure 3.7: Frequent Loop Analysis Tool

value of the SBB is a negative address offset. The *Frequent Loop Cache* (FLC) stores the execution frequency of each loop function at the index memory location that is based on the SBB value. A cache controller, called the *Frequent Loop Cache Controller*, keeps the data updated with the latest values. FLAT does not require the use of instrumentation code or any sampling techniques. Nonetheless, the accuracy of the loop detection relies on the size of the on-chip cache in the FPGA.

3.5.3 WoODSToCK

WOoDSTOCK [59] *(Watches Over Data STreaming On Computing element linKs)*, is a profiling tool that monitors the communication dataflow between Computing Processor Elements (CPEs) as shown in Figure 3.8

WOoDSToCK monitors the data flow between each CPE by adding monitors to the circuit which run in real time. The data link between each element of the system is created by *Fast Simplex Links* (FSLs) [71], available in Xilinx's MicroBlaze [73] soft-core processor. FSLs allow streaming and buffering of data between the hardware components of the system. The profiler utilizes the links to measure the stream of data between each CPE. It measures the number of run-time execution clock cycles to see which CPE is stalled or starved for data.

A stalled CPE occurs when a stream of data is at the input but little or no output data is coming out. A starved computing element occurs when little data is coming in

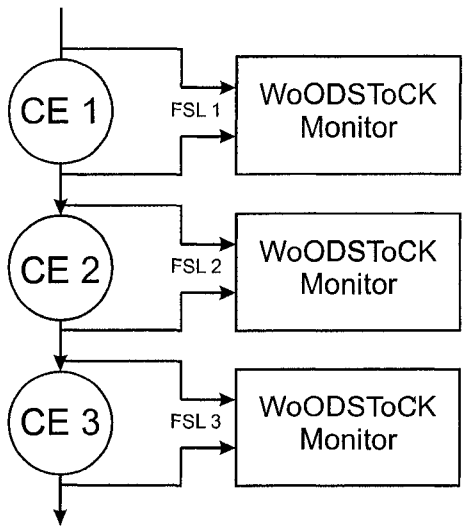

Figure 3.8: Watching Over Data Streaming on Computing Element Links

or going out of the CPE, but it is still running. The results obtained showed that the tool was able to detect bottlenecks using a pipelined system and a branching system benchmark.

3.6 Qualitative Comparison of Profiling Tools

There are a variety of profiling tools available today that can measure the performance of software code by collecting information about different performance metrics. The majority of these tools have one or more drawbacks related to accuracy, runtime overhead and extended execution time. Table 3.1 shows a comparison of the profiling tools discussed in this thesis.

Notice that SBP tools have functional and memory profiling capabilities. They do require the insertion of instrumentation code that is needed to interrupt the processor at specific intervals to sample the data stored in the hardware registers in the system. This can cause inaccurate profiled results to be reported along with the introduction

of performance overhead during execution and an increase in file size. This is not desirable in the design of embedded systems. One of the advantages of using simulators is that the original program does not require any instrumentation code. This is beneficial since this does not modify the behaviour of the software program, although simulating large programs is very slow and is therefore an impractical option for profiling large embedded system designs.

HCBP tools are mostly used for profiling memory systems, however, they do use techniques that are similar to those used by software-based profiling tools, such as sampling, which can affect the accuracy of the performance information retrieved. The accuracy of the profiled results is dependent on the frequency of the sampling rate.

FPGA-BP tools are clock-cycle accurate and do not introduce overhead during software execution. The software program may require minimal code disturbance or can be left alone, thus reducing the effect of unpredictable execution behaviour. As shown in the table, FPGA-BP tools are not restricted as are functional or memory profilers. They have the ability to detect communication bottlenecks between CPEs.

Feature	gprof	ISS	VTune	Valgrind	Purify	HWC	PMA	SnoopP	Woodstock	FLAT
Instrumentation Code	X		X	X	X					
Sampling	X		X	X	X	X	X			
Clock Cycle Accurate								X	X	X
Performance Overhead	X	X	X	X	X		X			
Simulation		X	X	X						
Software-Based	X	X	X	X	X					
Hardware-Based						X	X			
FPGA-Based								X	X	X
Functional Profiler	X	X	X					X	X	X
Memory Profiler				X	X	X	X			
Other Profiler									CPEs	

Table 3.1: Comparison of Profiling Tools

Chapter 4

The Airwolf Profiler

This chapter introduces the FPGA-Based Profiling tool, the *Airwolf* Profiler. The Airwolf Profiler contains a set of dedicated hardware counters that are used to profile software code running on the Nios II Processor. It is a System On Programmable Chip Builder (SOPC) Builder ready component [18] that can be instantiated on any Nios II Processor [32] based designs. The modification of the interface of the Airwolf Profiler can also be instantiated on other embedded soft-core processors such as Tensilica Xtensa Soft-Core Processor [8] and the Xilinx Microblaze Soft-Core Processor [73]. This chapter begins by describing the *Airwolf* Profiler's architecture. The later sections explain how each of *Airwolf's* segment counters measure time and the number of hits occurred. Finally a discussion of the *Airwolf* Profiler's software drivers used to profile software code is provided.

4.1 The Airwolf Architecture

The *Airwolf* Profiler is an on-chip FPGA-BP tool used to profile software programs running on the Nios II Processor in real-time. This is done by determining the run-time of each software function by accurately counting the number of system clock cycles. *Airwolf* does not require any instrumentation code added to the binary file. A pair of software drivers needs to be placed in between a software function block in the source code in order to activate and deactivate a particular profiling counter contained in *Airwolf*. This approach minimally disturbs the program and the software behaviour during execution. The goal of the *Airwolf* Profiler is to provide accurate results while minimally modifying the software code. Figure 4.1 depicts the *Airwolf* Profiler's Architecture.

As shown in the figure, the *Airwolf* Profiler contains the *Time Counter Enable* (TCE) module and 20 profiling counters. This is sufficient for profiling large programs that consist of a large number of software functions. Instantiating the *Airwolf* Profiler onto the Stratix EP1S40F780C5 FPGA [16] consumes 3,345 logic elements. The maximum operating frequency that the profiler can support is 120 MHz. Usually this frequency is used for high-speed Nios II Processor systems [32].

The TCE module contains 20 *Counter Enable* (CE) registers which are used to activate the appropriate profiling counter. The logic circuit in the TCE module is dependent on the *Address* and *Data_In* bus inputs that are being fed from the *Avalon Interface Bus* (AIB) [19]. The AIB contains all of the necessary control logic signals that are used to manipulate the CE registers in the TCE module. The accompanying software drivers of the *Airwolf* Profiler are programmed to access the appropriate CE register by sending a unique address onto the interfacing bus. The output of each CE register is fed into the input enable of the assigned profiling counter (shown as the *Time Counter Enabling Lines* (TCELs) in the figure).

The *Hits Counter Enable Lines* (HCEL) are the output control lines coming from

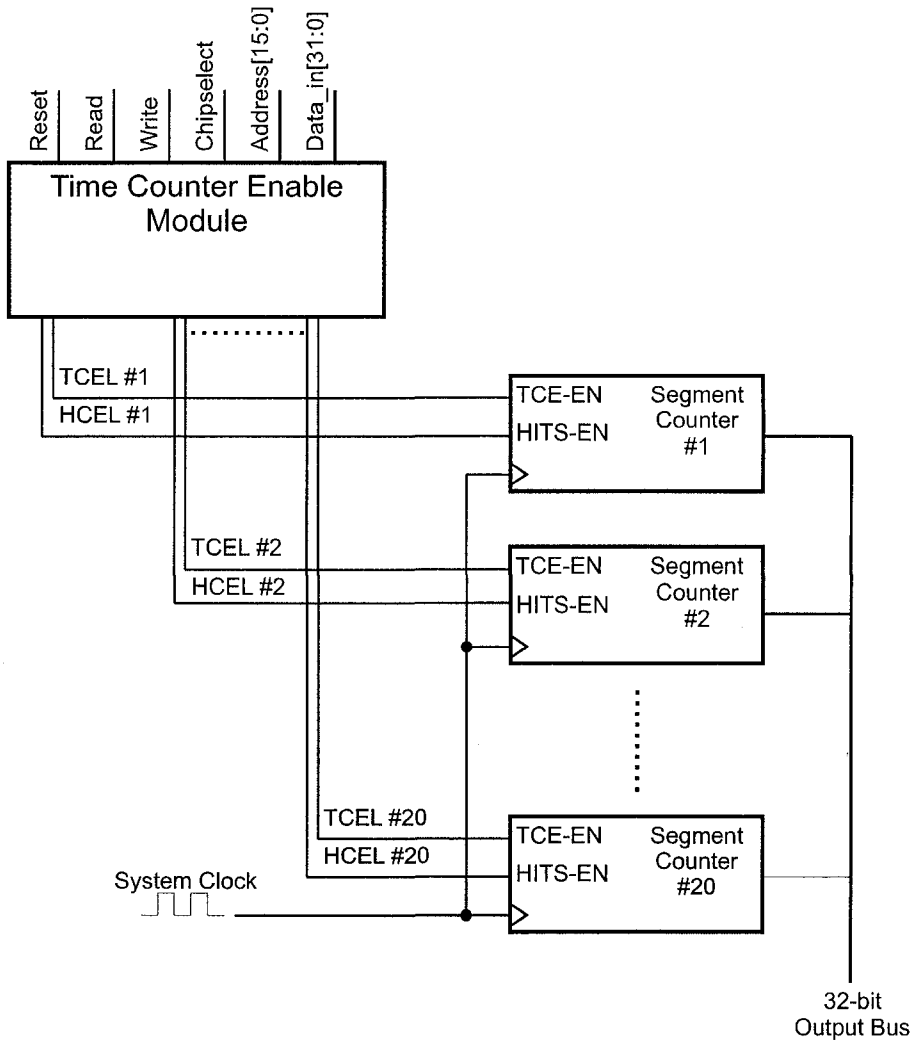

Figure 4.1: The Airwolf Profiler

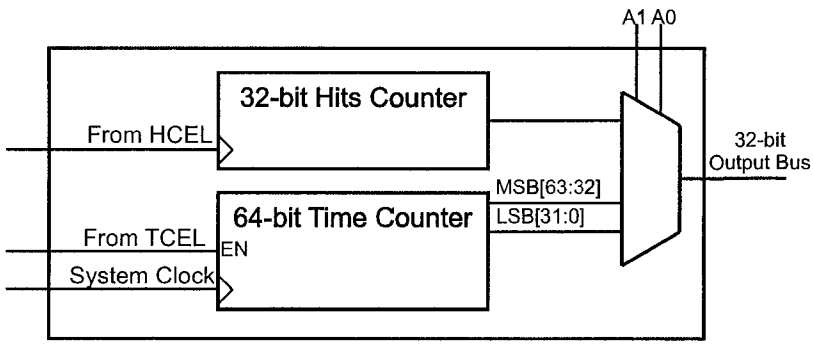

Figure 4.2: The Airwolf Profiling Counter

the TCE module. Their purpose is to indicate when a function has been called as the program is executing on the processor.

The *Data_In* and *Address* input buses are also used to extract the profiling data stored in the profiling counters. These data are sent out to a host computer through the *Data_Out* bus. A set of control signals provided by the AIB, namely the *chipselect*, *write_enable* and *read_enable* signals, are used to prevent any illegal input or output accesses of the CE registers and the profiling counters.

4.2 Airwolf Profiling Counter

The *Airwolf* Profiler contains 20 profiling counters which allow for up to 20 functions to be profiled at a time. Figure 4.2 depicts the contents of each profiling counter.

Each profiling counter actually consists of two counters, a 32-bit *hits counter* and a 64-bit *time counter*. The hits counter counts the number of positive edges of the input HCEL control signal. When the appropriate profiling software driver activates the profiling counter, the HCEL control signal becomes high for one clock cycle. This signifies that the assigned function has been activated and the hits counter is incremented by 1.

The 64-bit time counter is used to count the number of clock cycles of the cur-

rently executing function. Each 64-bit time counter is capable of measuring over 100 million hours of profiling time when using a 50 MHz system clock. This ensures that the overflow of the register will effectively never occur. There are two distinct inputs to each of the 64-bit time counters, which are the time counter enable and the clock inputs. The time counter enable input is fed by the appropriate TCEL control line, which controls the counting sequence of the counter. If the TCEL signal becomes high and remains at that state, the counter begins to count the number of positive edges of the system clock. If the TCEL signal becomes low, counting of the clock ticks is disabled. This concept is of great importance since *Airwolf* accurately counts the number of clock ticks a function has taken. This helps to provide accurate performance feedback which is beneficial for embedded system designers.

A multiplexer component that is controlled by the address bits from the *Address* input bus exists in every profiling counter. This mandates which data is assigned to the AIB. In the end, the profiled data stored in these counters will be extracted by calling the appropriate software driver and displayed back to the designer.

4.3 Airwolf's Software Drivers

To use the *Airwolf* Profiler, the source code must include the appropriate software drivers to control the counting of the profiling counters. There are 40 software drivers in total, and each profiling counter is assigned a pair of drivers. One driver is used to activate the appropriate profiling counter and is usually placed at the beginning of a function. Another driver is used to deactivate the appropriate profiling counter and is placed at the end of a function block. The sample code below illustrates this process.

The `AIRWOLF_SECTION_ONE_START()` driver calls on profiling counter #1 to start measuring by counting the number of clock ticks and the number of calls made to that function. Near the end of the function block, `AIRWOLF_SECTION_ONE_STOP()`

```
void somefunction (int n)
{
   AIRWOLF_SECTION_ONE_START();
   int addnumbers = 0;
   addnumbers += n;
   AIRWOLF_SECTION_ONE_STOP();
}

int main()
{
   AIRWOLF_RESET();
   somefunction (1000);
   AIRWOLF_OUTPUT();
   return (0);
}
```

Figure 4.3: An Example of Airwolf's Software Drivers

deactivates the CE #1 register which disables the profiling for profiling counter #1.

`AIRWOLF_RESET()` is a driver that resets and initializes all of the counters in the profiler to 0. This software driver is usually placed at the beginning of the `main` program.

`AIRWOLF_OUTPUT()` is a software function that extracts all of the data from the profiling counters in the *Airwolf* Profiler. The data stored in the 64-bit time counter needs to be split into two 32-bit words in order to be transported onto the 32-bit data bus. Initially, the Most Significant Word (MSW) is retrieved which corresponds to bits 32-63 of the 64-bit time counter. These bits are stored in a 32-bit variable. The Least Significant Word (LSW) is retrieved next and corresponds to bits 0-31 of the 64-bit time counter. Those bits are also stored in a separate 32-bit variable. To merge these data together, the 32-bit variable containing the MSW data is casted into a 64-bit variable and shifted 32 positions to the left. The 32-bit variable containing the LSW data bits is augmented with the 64-bit variable. This process is done for all

of the 20 profiling counters.

4.4 Summary

The *Airwolf* Profiler was introduced which describes the profiling architecture. Each profiling counter was presented which shows the type of collected and stored profiling information. In Chapter 5, a profiling environment is introduced, which is used to execute a set of software benchmarks. Each benchmark is profiled using the *Airwolf* Profiler. To determine the accuracy of the retrieved performance information provided by the *Airwolf* Profiler, the results are compared against a well-known software based profiler, GNU's *gprof*. In addition, performance overhead analysis is conducted, which compares the run-time for each function that was compiled with and without instrumentation code.

Chapter 5

Experimental Results

This chapter presents analysis and comparison of the profiled results obtained by using *Nios2-gprof* (SBP tool) and the *Airwolf* (FPGA-BP tool) Profilers. We first describe the experimental environment and the profiling software benchmarks used. This discussion includes details of the instantiated components of the Nios II Processor System and a brief description of the operations involved in each of the software benchmarks. A thorough analysis and critical comparisons of profiling results is presented. Finally, a performance overhead analysis is presented, which compares the execution times of each software function with and without instrumentation code.

5.1 The Nios II Profiling Environment

For this experiment, a Nios II Processor system was created to serve as the profiling environment for the software benchmarks. This environment consists of a processor core, system timers, memory and a bus interconnect which connects all the instan-

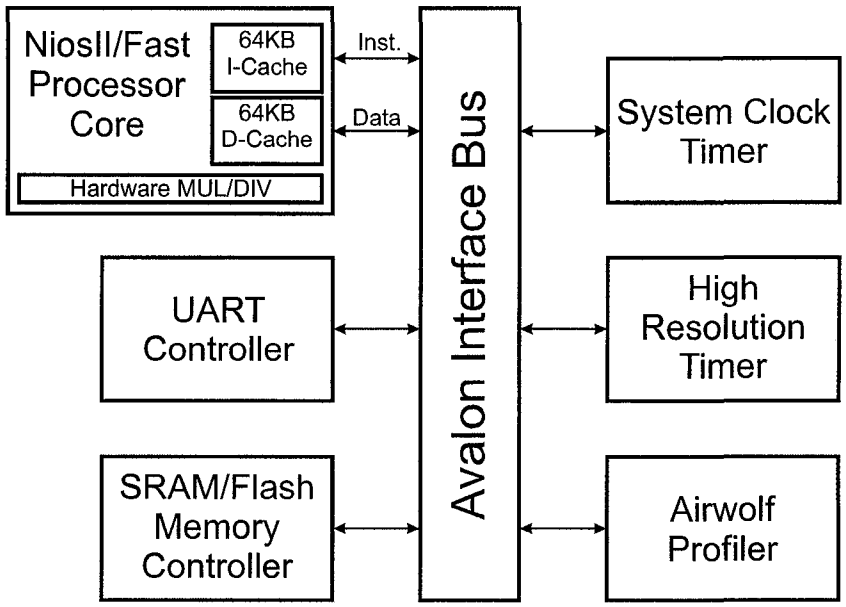

Figure 5.1: The Nios II Profiling Environment

Nios Development Board	
Stratix Professional Edition	41250 Logic Elements
Static RAM (Off-Chip) Module	1MB
Flash Ram (Off-Chip) Module	8MB

Table 5.1: Nios Development Board Components

tiated components. Figure 5.1 depicts the Nios II Processor System, which will be referred to as the *Nios II Profiling Environment* (Nios-II-PE).

Table 5.1 lists the instantiated components used in the Nios-II-PE. The Nios-II-PE consists of the fast version of the Nios II Processor core, which is a soft-core processor that is optimized for high performance in computationally-intensive applications at the expense of consuming more logic elements on an FPGA. This processor is suitable for executing the benchmarks used in this experiment. The core contains multiply and divide hardware accelerators which allow multiplication and division operations

to be executed in hardware [32]. In addition, it contains separate instruction and data cache memories, each having 64KB. For the program, stack and data memories, the Nios-II-PE utilizes the 1 MB static Random Access Memory (RAM) module which is located off-chip. Software benchmarks are downloaded onto this memory module. There are two timers in this system, namely the system clock and high performance timers. They are required for *Nios2-gprof* in order to measure the run-time of the software functions and by some of the software benchmarks as well. An instance of the *Airwolf* Profiler is used in the Nios-II-PE, consisting of all 20 profiling counters. Each of these counters is assigned a specific software function to profile. Software function assignments are based on the placement of the software drivers in the program, as was explained in Section 4.3. The Universal Asynchronous Receiver and Transmitter (UART) controller is used to communicate with the Nios-II-PE and to transfer streaming messages back to the host computer. All of the instantiated components in the Nios-II-PE are connected using the Avalon Interface Bus [19] which provides all of the necessary control logic and data signals that are used to communicate between each instantiated component.

5.2 FPGA Development Board and Design CAD Tools

The Nios Development Kit [3] was used to implement the Nios-II-PE. This kit contains a Nios Development Board, Stratix Professional Edition, featuring a Stratix EP1S40F780C5 FPGA chip. The chip features 41,250 logic elements, 3,423,744 memory bits and 14 Digital Signal Processing (DSP) blocks [22]. There are available off-chip memory modules that can be used, which include the 8MB flash memory, the 1MB SRAM and the 16MB SDRAM modules. In this experiment, the 1MB SRAM was used for the program, stack and data memories for each benchmark. All of the

components on the development board utilized the 50MHz clock oscillator as the system clock of the Nios-II-PE.

The supporting CAD tools that were used in this experiment are Quartus II Version 5.0 SP2 [20], System On Programmable Chip (SOPC) Builder Version 5.0 [23] and Nios II Integrated Development Environment (IDE) Version 5.0 [21].

Quartus II [20] is a design environment that is used to synthesize hardware description language (HDL) files and to generate a Static-RAM Object Files (SOF) that are used to program the FPGA. SOPC Builder [23] is a system builder tool that builds embedded systems using different instantiated IP cores. It automatically generates HDL files based on the instantiated components that are used in the system. In addition, user-specified IP cores can be imported into SOPC Builder and can also be utilized in a system.

Nios II IDE [21] is an environment that is used to generate and compile C and C++ code and download its binary image to run on a Nios II Processor System. It contains a number of debugging tools that the designer may use to debug software code, enabling them to view the data contents inside the Nios II Processor core's registers. It also comes with an interface that is used to communicate with the Nios II Processor system over a serial cable that is connected to the FPGA development board. The console window that is displayed on the host computer shows the output generated by the Nios II Processor System and other status messages.

5.3 Profiling Tools Setting

The profiling tools used in this experiment are *NiosII-gprof* and the *Airwolf* Profiler. Each benchmark was imported into the Nios II IDE. There were some additional settings that were applied to the software benchmarks in order to utilize these profiling tools:

- *Nios2-gprof*: To utilize this profiler, the original program must be compiled with instrumentation code (`-pg`) which causes the GCC compiler to insert extra software interrupts and variable counters into the program's binary file. This is required by *Nios2-gprof* so that it can collect performance information during the execution of the software program.

- *Airwolf* Profiler: The *Airwolf* Profiler requires a pair of software drivers added to the source code of the program. One driver is used to activate and the other to deactivate the assigned function's profiling counter. This ensures that the reported execution time is dedicated to the assigned function.

Each benchmark was compiled using the *Nios II GCC* compiler by applying the highest optimal compilation (`-O3`) setting. The compiler generates the executable binary by optimizing the code for fast performance at the expense of a slightly larger file size [1].

5.4 Profiling Software Benchmarks

The software benchmarks used in this experiment are listed in Table 5.2. These benchmarks were based on the embedded software benchmarks suite *MiBench* [43, 2] and the *UTNiosbenchmarks* [53]. The following paragraphs describe each benchmark briefly.

- *BitCount*: This benchmark tests the bit manipulation capabilities of a microprocessor. Inputs to this benchmark are arrays of 1s and 0s. *BitCount* uses five bit-counting and manipulation algorithms which are the following: *optimized 1-bit per loop, recursive bit count by nibbles, non-recursive bit count by nibbles using a table look-up, non-recursive bit count by bytes using table look-up* and *shift and count bits* [43]. This algorithm was executed for 10,000,000 iterations.

Profiling Software Benchmarks	
BitCount	Performs several bit manipulations for 10,000,000 iterations
Dijkstra	Computes the shortest path between 160 nodes
Game of Life	Cellular automation program run for 100,000 passes
Fibo_Matrix_Mult	Computes the 40th Fibonacci term and then multiplies two 250x250 matricies
Dhrystone	Tests the integer performance of a processor for 100,000,000 iterations

Table 5.2: Benchmark Descriptions

- *Dijkstra*: This algorithm, developed by Edsgar W. Dijkstra, finds the shortest path between any pair of nodes. *Dijkstra* uses an adjacency matrix to compute the shortest distance that is represented by a 100x100 matrix [43]. This benchmark was modified to find the distance between 160 distinctive nodes.

- *Game of Life*: Based on John Conway's game of life, [15], this benchmark is a cellular automation program which models a cell that is initially alive or dead dependent on the seed configuration [61]. A set of rules are followed which determine the cell's birth or death in the next generation cycle. This benchmark was executed for 100,000 passes.

- *Fibo_Matrix_Mult*: There are two functions in this benchmark that are called sequentially. The first function is `Fibonacci` which computes the 40th term of a Fibonacci sequence recursively. The second function is `Matrix_Mult` which multiplies two 250x250 matrices.

Nios2-gprof				Airwolf Profiler			
FCN Name	Time (Secs)	% Time	# of Calls	FCN Name	Time (Secs)	% Time	# of Calls
Dijkstra	41.56	71.43	160	Dijkstra	42.27	70.98	160
Enqueue	16.27	27.96	192739	Enqueue	16.61	27.89	192739
Dequeue	0.25	0.43	192739	Dequeue	0.52	0.88	192739
Read_int	0.05	0.09	25600	Read_int	0.12	0.20	25600
Qcount	0.05	0.09	192899	Qcount	0.03	0.05	192899

Table 5.3: Profiled Results for Dijkstra

- *Dhrystone*: A synthetic benchmark which assesses a system's integer performance. The Nios II IDE provided this program to measure the performance of the Nios II Processor Core [10].

5.5 Comparison of Profiled Results

Each benchmark was executed with *Nios2-gprof* and the *Airwolf* Profiler with their respective software compilation settings. In the subsequent paragraphs, an analysis of the profiled results is presented for each of the benchmarks listed in Table 5.2.

5.5.1 Dijkstra

Table 5.3 shows the profiled results for the *Dijkstra* benchmark. The first four columns show the results obtained by *Nios2-gprof* and and the latter four columns show the results obtained with *Airwolf* profiler. The first column gives the function name. The second column shows the execution time of each function. The third column shows the function's execution as a percentage of total execution time of the benchmark. The number of function calls is displayed in the fourth column. The same explanation

Nios2-gprof				Airwolf Profiler			
FCN Name	Time (Secs)	% Time	# of Calls	FCN Name	Time (Secs)	% Time	# of Calls
Fibonacci	172.14	82.69	204668308	Fibonacci	195.17	84.46	204668309
Matrix_Mult	36.03	17.31	1	Matrix_Mult	35.90	15.54	1

Table 5.4: Profiled Results for Fibo_Matrix_Mult

applies for the remaining columns in the table and for all subsequent tables.

Each profiler's results are alike, having similar execution times and rankings of computationally intensive functions. The `Dijkstra` function is reported to run for 41.56 seconds by *Nios2-gprof* whereas the *Airwolf* Profiler reported 42.27 seconds.

There are very minor differences in the reported execution times of the remaining software functions. This implies that *Nios2-gprof* reports results with comparable accuracy to those of the *Airwolf* profiler for smaller, less computationally intensive benchmarks. *Airwolf* attained an improvement in accuracy of 1.67%.

5.5.2 Fibo_Matrix_Mult

Table 5.4 depicts the profiled results for the *Fibo_Matrix_Mult* benchmark. *Nios2-gprof* reported that the `Fibonacci` function was called 204,668,309. Similarly, *Airwolf* reported that the number of calls to `Fibonacci` was 204,668,309 times. In terms of the run-time, *Nios2-gprof* and *Airwolf* reported that the function was running for 172.14 and 195.17 seconds respectively. This implies that the sampling technique used in *Nios2-gprof* has produced an inaccurate report of the execution time when profiling recursive function calls. In conrast, the clock-cycle counting method that *Airwolf* utilizes shows an 11.79% accuracy improvement in the reported time for that function.

The `Matrix_Mult` function had very minor difference in the reported time between

Nios2-gprof			
FCN Name	Time (Sec)	% Time	# of calls
set_new_grid_pres_state	24.02	30.61	100000
set_cell_next_state	21.92	27.93	20000000
adjust_neigh_cnt	19.70	25.11	20000200
set_grid_next_state	12.57	16.02	100000
init_grid	0.26	0.33	1

Table 5.5: Profiled Results for Game for Life using *Nios2-gprof*

the two profilers. The percentage difference is 0.36%.

5.5.3 Game of Life

Tables 5.5 and 5.6 shows the results for the *Game of Life* benchmark using *Nios2-gprof* and *Airwolf* respectively. *Nios2-gprof* reported the function `set_new_grid_pres_state` as being the longest running function. This is reported similarly by *Airwolf* as well. Looking further into the table, the computationally intensive functions are ranked differently between the two profilers. *Nios2-gprof* ranked `set_cell_next_state`, `adjust_neigh_cnt`, and `set_grid_next_state` in the order of the longest running functions.

Airwolf had a different ranking which listed `set_grid_next_state`, `set_cell_next_state` and `adjust_neigh_cnt` as the order of computationally intensive functions. Results like those reported by *Nios2-gprof* can potentially mislead embedded designers to assign a function for hardware implementation.

The reported times of each function as reported by *Nios2-gprof* are slightly inaccurate. More noticeably is the `set_grid_next_state` function, which was reported to have run for 12.57 and 18.57 seconds by *Nios2-gprof* and *Airwolf* respectively. Using *Airwolf* can provide an increase in accuracy of 32.3%.

Airwolf Profiler			
FCN Name	Time (Sec)	% Time	# of calls
set_new_grid_pres_state	28.99	36.32	100000
set_grid_next_state	18.57	23.28	100000
set_cell_next_state	17.62	22.08	20000000
adjust_neigh_cnt	14.58	18.28	20000200
init_grid	0.00021	0.00036	1

Table 5.6: Profiled Results for Game for Life using *Airwolf*

Nios2-gprof			
FCN Name	Time (Secs)	% Time	# of Calls
ntbl_bitcnt	71.88	22.35	80000000
bit_shifter	66.27	20.60	10000000
bit_count	63.55	19.76	10000000
main	47.10	14.64	1
ntbl_bitcount	24.51	7.62	10000000
ar_btbl_bitcount	19.76	6.14	10000000
bitcount	17.41	5.41	10000000
btbl_bitcnt	6.47	2.01	
bw_btbl_bitcount	4.40	1.37	10000000
Flipbit	0.28	0.09	

Table 5.7: Profiled Results for BitCount using *Nios2-gprof*

Airwolf			
FCN Name	Time (Secs)	% Time	# of Calls
bit_shifter	196.64	54.40	10000000
bit_count	61.98	17.15	10000000
ntbl_bitcnt	51.34	14.20	80000000
ntbl_bitcount	22.26	6.16	10000000
ar_btbl_bitcount	15.04	4.16	10000000
bitcount	12.63	3.49	10000000
bw_btbl_bitcount	4.40	0.44	10000000
main	0.01	0.00	1
btbl_bitcnt	0.00	0.000	
Flipbit	0.00	0.00	

Table 5.8: Profiled Results for BitCount using *Airwolf*

5.5.4 BitCount

Tables 5.7 and 5.8 shows the profiled results for the *BitCount* benchmark using *Nios2-gprof* and *Airwolf* profilers respectively. There is a significant difference in the reported execution time of each function when the results from each profiler are compared. Not only are the execution times different, but *Nios2-gprof* also ranked the most time consuming functions differently than *Airwolf*. *Nios2-gprof* listed the `ntbl_bitcnt`, `bit_shifter` and `bit_count` as the most time consuming functions, whereas the *Airwolf* Profiler reported that the `bit_shifter`, `bit_count` and `ntbl_bitcnt` functions contributed the most toward the total execution time of the benchmark.

Nios2-gprof reported that `bit_shifter` ran for 66.27 seconds whereas *Airwolf* Profiler has measured that function to take 196.64 seconds on the processor. Once

again, due to the sampling technique used by *Nios2-gprof*, the profiler provided an inaccurate reporting of the execution time. *Airwolf* Profiler provided up to 66.2% improvement in accuracy in some of the functions.

As for the `ntbl_bitcnt` function, which was called recursively, *Nios2-gprof* and *Airwolf* reported that the function was running for 71.88 and 51.34 seconds respectively. This shows that *Nios2-gprof* reports inaccurate execution times when profiling recursive functions.

Nios2-gprof reported that the `btbl_bitcnt` and `Flipbit` functions were called during the execution of the benchmark. However, the *Airwolf* Profiler did not detect calls to those functions. The insertion of instrumentation code not only generates additional function calls and interrupts, but it can also cause unpredictable behaviour of the executing program.

5.5.5 Dhrystone

Table 5.9 shows the profiled results for the *Dhrystone* benchmark. Both profilers have similarly ranked the most time consuming functions. However, the reported execution times of each function were quite different. `Proc_8` was reported to run for 100.52 seconds by *Nios2-gprof* whereas *Airwolf* reported 78.26 seconds. This shows a 22.1% in improvement with the *Airwolf* profiler. Additionally `Proc_6` was reported by *Nios2-gprof* to run for 80.52 seconds and *Airwolf* reported that function to run for 62.28 seconds. The improved accuracy using *Airwolf* in that function is 22.6%. `Proc_4` also had a significant difference in the reported execution time. *Nios2-gprof* reported `Proc_4` was running at 30.01. In contrast, *Airwolf* reported that function was running at 18.00 seconds which this provides a 40% accuracy improvement.

Another noticeable inaccurate reporting of the execution times are the functions `Proc_1`, `Proc_3` and `Func_1`. `Proc_1` was reported to take 131.84 and 106.53 seconds by *Nios2-gprof* and the *Airwolf* Profiler respectively. This amounts to a 19.19%

Nios2-gprof				Airwolf Profiler			
FCN Name	Time (Secs)	% Time	# of Calls	FCN Name	Time (Secs)	% Time	# of Calls
Func_2	240.02	30.02	100000000	Func_2	253.64	38.32	100000000
Proc_1	131.84	16.49	100000000	Proc_1	106.53	16.01	100000000
Proc_8	100.52	12.57	100000000	Proc_8	78.26	11.83	100000000
Proc_6	80.52	10.07	100000000	Proc_6	62.28	9.41	100000000
Func_1	67.69	8.47	300000000	Proc_2	36.00	5.44	100000000
Proc_3	49.19	6.15	100000000	Func_1	34.69	5.24	300000000
Proc_7	38.13	4.77	300000000	Proc_7	30.14	4.55	300000000
Proc_2	36.91	4.62	100000000	Proc_3	22.13	3.34	100000000
Proc_4	30.01	3.75	100000000	Proc_4	18.00	2.72	100000000
Proc_5	15.11	1.89	100000000	Func_3	10.14	1.53	100000000
Func_3	9.69	1.21	100000000	Proc_5	10.00	1.51	100000000

Table 5.9: Profiled Results for Dhrystone

improvement in accuracy when using the *Airwolf* Profiler. Another observation is with regards to the reported times of `Func_1` and `Proc_3`. *Nios2-gprof* reported that `Func_1` took 67.69 seconds to execute and `Proc_3` ran for 49.19 seconds. However, the results obtained with the *Airwolf* Profiler showed that `Func_1` had an execution time of 34.69 and that `Proc_3` executed for 22.13 seconds. This amounts to a 55% improvement in accuracy with the *Airwolf* Profiler.

5.5.6 Summary

The *Airwolf* Profiler has experimentally demonstrated a significant improvement in achieving accurate profiled results. *Airwolf's* measuring technique is to precisely count the number of system clock ticks of a function has taken without any sam-

pling methods or instrumentation code inserted. In some of the profiling software benchmarks, *Airwolf* has attained 66.2% improvement. In addition, *Airwolf* has ranked computationally intensive functions differently than the software-based profiler, *Nios2-gprof*. These improvements can greatly benefit designers and guides them in making a proper hardware-software partition of the embedded system. In the next section, performance overhead analysis is conducted which compares the actual runtime of a program with and without the insertion of instrumentation code into the software program.

5.6 Performance Overhead Analysis

Nios2-gprof requires the C/C++ file to be compiled with instrumentation code which generates additional software interrupts and counter variables in the original program. This can lead to a large increase in the execution time of the benchmark and can cause an inconvenience to the embedded system designer who has to wait (potentially, for many hours) to retrieve the profiled results. This especially applies as the software code size grows larger.

In this section, an analysis of the performance overhead will be conducted for the software benchmarks discussed above. Each software program was compiled with the default debug (-g) setting while the same assigned functions were profiled with the *Airwolf* Profiler. The performance overhead was determined by summing the execution time of each profile run, with and without instrumentation code.

5.6.1 Dijkstra

Table 5.10 shows the overhead performance analysis for *Dijkstra*. Column 1 lists the function names. Columns 2 and 3 show the execution times when the program was executing with and without instrumentation code respectively. The last column

FCN Name	Time (Sec) No Instrumentation	Time (Sec) with instrumentation	Difference (Sec)
Dijkstra	42.20	43.24	1.04
Enqueue	16.60	17.22	0.62
Dequeue	0.52	0.92	0.4
Read_int	0.12	0.12	0
Qcount	0.031	0.031	0
Performance Overhead: 3.35%			

Table 5.10: Performance Overhead Analysis for Dijkstra

FCN Name	Time (Sec) No Instrumentation	Time (Sec) with instrumentation	Difference (Sec)
Fibonacci	195.17	357.90	162.74
Matrix_Mult	35.90	36.00	0.10
Performance Overhead: 41.34%			

Table 5.11: Performance Overhead Analysis for Fibo_Matrix_Mult

shows the time difference between the two compilation runs.

As evident from the table, there is very little time difference when instrumentation code is added, at most 1.04 seconds. This implies that profiling with *Nios2-gprof* on smaller benchmarks, such as *Dijkstra*, contributes minimal performance overhead. In this case, only 3.35% of additional execution time was contributed by the instrumentation code.

5.6.2 Fibo_Matrix_Mult

Table 5.11 depicts the performance overhead analysis for the *Fibo_Matrix_Mult* benchmark. Notice that the `Fibonacci` function has taken 162.74 seconds of additional

FCN Name	Time (Sec) No Instrumentation	Time (Sec) with instrumentation	Difference (Sec)
set_new_grid_pres_state	28.98	42.71	13.73
set_grid_next_state	18.57	32.28	13.70
set_cell_next_state	17.62	17.60	0.02
adjust_neigh_cnt	14.58	14.61	0.03
init_grid	0.00021	0.00021	0.00
Performance Overhead: 25.60%			

Table 5.12: Performance Overhead Analysis for Game of Life

execution time. The added instrumentation code changed the behaviour of the software benchmark. Since the `Fibonacci` function was called recursively, this implies that instrumentation code adds significant performance overhead when profiling recursive functions with *Nios2-gprof*. This has caused the entire benchmark to have a performance overhead of 41.34%.

5.6.3 Game of Life

Table 5.12 shows the performance overhead analysis for the *Game of Life* benchmark. The `set_new_grid_pres_state` and `set_grid_next_state` functions show noticeable increases in execution time with the added instrumentation code by 13.73 and 13.70 seconds respectively. The remainder of the functions had very minor differences, at most 0.033 seconds. Once again, the inserted code caused an increase in run-time of those two functions, contributing nearly 25.6% of performance overhead.

5.6.4 BitCount

Table 5.13 demonstrates the performance overhead analysis for the *BitCount* benchmark. The instrumentation code added an additional 48.43 seconds in execution time

FCN Name	Time (Sec) No instrumentation	Time (Sec) with instrumentation	Difference (Sec)
bit_shifter	196.64	197.31	0.67
bit_count	61.98	62.18	0.20
ntbl_bitcnt	51.34	99.77	48.43
ntbl_bitcount	22.26	22.31	0.05
ar_btbl_bitcount	15.04	15.09	0.05
bitcount	12.63	12.66	0.03
bw_btbl_bitcount	1.60	1.60	0.00
btbl_bitcnt	0.00	0.00	0
Performance Overhead: 12.10%			

Table 5.13: Performance Overhead Analysis for BitCount

to the recursively called function `ntbl_bitcount`. This strongly supports the idea that profiling recursive functions with *Nios2-gprof* can cause a significant increase in run-time execution. The other functions listed in this table had very little effect in the execution time. This has resulted an overall performance overhead of 12.10%.

5.6.5 Dhrystone

Table 5.14 depicts the execution time differences of each software function in *Dhrystone*. Some of the functions showed a slight decrease in execution time, resulting in the negative time differences shown in the table. The instrumentation code may have caused a change in behaviour in those functions. Since those negative values are diminutive however, it minimally affects the entire benchmark's execution time. Notice that the software functions `Func_2` and `Proc_5` show a significant increase in run-time, adding 107.59 and 67.73 seconds respectively. The overall performance overhead for *Dhrystone* is 21.59% when using *Nios2-gprof*.

FCN Name	Time (Sec) No instrumentation	Time (Sec) with instrumentation	Difference (Sec)
Func_2	253.64	361.23	107.591
Proc_1	106.53	106.00	-0.531
Proc_8	78.26	83.00	4.736
Proc_6	62.28	130.00	67.725
Proc_2	36.00	37.59	1.590
Func_1	34.69	33.00	-1.687
Proc_7	30.14	30.00	-0.138
Proc_3	22.13	25.00	2.868
Proc_4	18.00	18.25	0.247
Func_3	10.14	10.00	-0.140
Proc_5	10.00	10.00	0.000
Performance Overhead: 21.59%			

Table 5.14: Performance Overhead Analysis for Dhrystone

5.6.6 Summary

The results presented in this analysis have shown that the insertion of the instrumentation code in the program's binary file contributed to additional and unnecessary run-time for certain software functions. In particular the computationally intensive functions executed longer than normal, contributing up to 41.30% of performance overhead. The instrumentation code not only adds additional interrupt calls but has changed the behaviour of the entire program execution. This is undesirable since designers must rely on the actual program behaviour in order to retrieve the accurate profiled results. FPGA-BP tools require minimal or no instrumentation code added to the program which makes them more desirable compared to SBP tools.

Chapter 6

Conclusions and Future Work

This dissertation has discussed and qualitatively compared the existing profiling tools used for profiling software code. The different measuring techniques that each profiler uses can retrieve different performance metrics, although with varying accuracy in the profiled results. A proposed FPGA-based profiler, the *Airwolf* Profiler, was used to profile a set of profiling software benchmarks. These results were compared with those generated by a well-known software-based profiler, *Nios2-gprof*. The results show that FPGA-based profilers provide a significant improvement in accuracy of the profiled results based on the measured execution time of each software function. This benefits embedded designers and guides them to a proper hardware-software partition of an embedded system. This chapter gives a brief summary of the work that has been presented.

Chapter 2 described the four different approaches for the design of embedded systems: the *Traditional Design Methodology*, *Hardware-Software Co-Design*, *Function-Architecture Co-Design* and *Platform-Based Design*.

In Chapter 3, a comprehensive survey and comparison of existing profiling tools was presented. Proposed classification of these tools was made, namely *Software-Based Profilers* (SBP), *Software-Based Memory Profilers* (SBMP), *Hardware Counter-Based Profilers* (HCBP) and *FPGA-based Profilers* (FPGA-BP).

In Chapter 4, a FPGA-BP tool, the *Airwolf* Profiler, was introduced. *Airwolf's* profiling architecture was discussed and a description of how the profiler accurately measures the execution time of a software function was given. *Airwolf's* profiling counters along with its supporting software drivers were also presented.

In Chapter 5, the profiling environment and the supporting CAD tools were explained. The profiling software benchmarks were described, which were used to obtain profiled results using *Nios2-gprof* and the *Airwolf* profilers. An analysis of the retrieved profiled results from both profilers was presented. This analysis was based on the execution time that each profiler measured. It was experimentally demonstrated that the *Airwolf* Profiler provided up to a 66.2% improvement in accuracy over *Nios2-gprof* in some of the software functions. In addition, performance overhead analysis was used to compare the execution times between two programs: one that contained instrumentation code and one that did not. It was shown that the insertion of instrumentation code caused a significant increase in execution time in some of the functions, contributing up to 41.34% in run-time performance overhead. This added time and overhead is unnecessary since it causes *Nios2-gprof* to report inaccurate execution times of each function and causes delays for the designer to retrieve the profiled results.

6.1 Research Contributions

The research contributions made in this dissertation are as follows:

- An FPGA-BP tool, the *Airwolf* Profiler, was proposed and developed to profile

a set of profiling software benchmarks. It has provided highly accurate profiled results which are very useful for embedded system designers.

- The *Nios II Profiling Environment* was developed and was used to implement the two profilers in order to execute and profile different software benchmarks.
- Performance overhead analysis was conducted in order to observe the effects of adding instrumentation code to a program's binary file. It was shown that certain software functions executed abnormally, causing an increase in run-time execution.

6.2 Future Work

The *Airwolf* Profiler was designed for research purposes to profile software applications running on an Altera Nios II Processor [32] implemented on an Altera Stratix FPGA [22]. The tool can easily be modified to become an instruction address-based profiler that has the capability of monitoring the current instruction in execution on the processor. This concept can provide an improvement in the profiled results compared to the current software driver strategy.

In future work, the *Airwolf* Profiler can be enhanced to cover memory profiling as well, so that it can monitor memory related events such as the number of off-chip memory accesses, cache misses and memory leakages. This can further benefit embedded system designers and help in improving certain portions of the software code that cause memory related performance issues. The *Airwolf* profiler can also be easily modified to work with other FPGA-based soft core processors such as Xilinx MicroBlaze [73].

References

[1] GNU GCC Document. http://gcc.gnu.org/onlinedocs/gcc-3.2.3/gcc, Accessed February 2006.

[2] MiBench Version 1.0. http://www.eecs.umich.edu/mibench/, Accessed February 2006.

[3] Nios II Development Kit. http://www.altera.com/products/devkits/altera/kit-nios_1S40.html, Accessed September 2005.

[4] Sun One Studio 5, Standard Edition. http://www.sun.com/download/products.xml?id=3edd36bd, Accessed September 2005.

[5] The Linux Homepage. http://www.linux.org, Accessed September 2006.

[6] The Unix System. http://www.unix.org, Accessed September 2006.

[7] The Windows Homepage. http://www.microsoft.com/windows/default.asp, Accessed September 2006.

[8] The Xtensa 7 Processor for SOC Design. http://www.tensilica.com/products/xtensa_7.htm, Accessed September 2006.

[9] *AMD Athlon Processor, x86 Code Optimization Guide*, 2002.

[10] Dhrystone Code. http://www.netlib.org/benchmark/dhry-c, Accessed May 2005.

[11] Intel Corporation. http://www.intel.com, Accessed May 2005.

[12] *Mentor Graphics Performance Profiler User's and Reference Manual (Software Version 2.2)*, May 2005.

[13] SystemC. http://www.systemc.org, Accessed July 2005.

[14] Valgrind. http://www.valgrind.org, Accessed October 2005.

[15] John Conway's Game of Life. http://www.bitstorm.org/gameoflife/, Accessed May 2006.

[16] Altera Corporation. *Nios Development Board Reference Manual, Stratix Professional Edition*, September 2004.

[17] Altera Corporation. http://www.altera.com/, Accessed January 2005.

[18] Altera Corporation. *Altera Embedded Peripherals*, October 2005.

[19] Altera Corporation. *Avalon Interface Specification*, April 2005.

[20] Altera Corporation. *Introduction to Quartus II Version 5.0*, April 2005.

[21] Altera Corporation. *Nios II IDE Help System Version 5.0*, October 2005.

[22] Altera Corporation. *Stratix Device Handbook - Volume 1*, July 2005.

[23] Altera Corporation. *System On Programmable Chip Builder Version 5.0*, October 2005.

[24] D. L. Anderson and D. Brucks. An introduction to sampling and time. Technical report, Intel Corporation, 2005.

[25] K. Banovic, M.A.S. Khalid, and E. Abdel-Raheem. Fpga-based rapid prototyping of digital signal processing systems. In *Proc. of the 48th Mid-West Symposium on Circuits and Systems*, pages 647–650, August 2005.

[26] A. Bonivento and A. Sangiovanni-Vincentelli. Platform based design for wireless sensor networks. In *Proc. of the 2nd Annual Workshop on Networking with Ultra Wide Band and Workshop on Ultra Wide Band for Sensor*, pages 9–19, July 2005.

[27] S. Brini, D. Benjelloun, and F. Castanier. A flexible virtual platform for computational and communication architecture exploration of dmt, vdsl modems. In *Proc. of the 2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 164–169, December 2003.

[28] S. Brown, S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform-infrastructure for application performance tuning using hardware-counters. In *Proc. of the 2000 ACM/IEEE conference on Supercomputing*, 2000.

[29] D. Burger and T. M. Austin. The simplescalar tool set version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[30] C. J. N. Coelho Jr., D. C. Da Silva Jr., and A. O. Fernandes. Hardware-software codesign of embedded systems. In *Proc. of the 11th Brazilian Symposium on Integrated Circuit Design*, pages 2–8, January 1998.

[31] K. Compton and S. Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, June 2002.

[32] Altera Corporation. *Nios II Processor Handbook*, October 2005.

[33] C. Erickson. Memory leak detection in c++. *Linux Journal*, 2002, October 2002.

[34] C. Erickson. Memory leak detection in embedded systems. *Linux Journal*, 2003, June 2003.

[35] R. Ernst, J. Henkel, and T. Benner. Hardware-sotware cosynthesis for micro-controllers. *IEEE Transactions on Design and Test of Computers*, 10(4):64–75, December 1993.

[36] J. Fenlason and R. Stallman. Gnu gprof. http://www.gnu.org/software/binutils/manual/gprof-2.9.1/, January 1997.

[37] J. Fleishmann and K. Buchenrieder. A hardware-software prototyping environment for dynamically reconfigurable embedded systems. In *Proc. of the 6th International Workshop on hardware-Software Co-Design*, pages 105–109, March 1998.

[38] J. Fleishmann and K. Buchenrieder. Codesign of embedded systems based on java and reconfigurable hardware components. In *Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, pages 768–769, March 1999.

[39] D. W. Franke and M. K. Purvis. Hardware/software codesign: a perspective. In *Proc. of the 13th international conference on Software engineering*, pages 344–352, May 1991.

[40] A. Gordon-Ross and F. Vahid. Frequent loop detection using efficient non-intrusive on-chip hardware. In *Proc. of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 117–124, November 2003.

[41] Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A quantitative analysis of the speedup factors of fpgas over processors. In *Proc. of the 12th International Symposium on Field Programmable Gate Arrays*, pages 162–170, February 2004.

[42] R. K. Gupta and G. DeMicheli. Hardware-software cosynthesis for digital systems. *IEEE Transactions on Design and Test of Computers*, 10:29–41, September 1993.

[43] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. D. Brown. Mibench: A free, commercial representative embedded benchmark suite. In *Proc. of the 4th Annual Workshop on Workload Characterization*, pages 3–14, December 2001.

[44] IBM Corporation. *Rational PurifyPlus, Rational Purify, Rational Pure Coverage, Rational Quantify. Installing and Getting Started. Version 2003.06.00, Technical White Paper.*

[45] Intel Corporation. *Using Intel VTune's Counter Monitor*, January 2005.

[46] Intel Corporation. IA-32 Intel Architecture Software Developer's Manual. http://developers.sun.com/prodtech/cc/articles/pcounters.html, Accessed February 2006.

[47] Intel's VTune. http://www.intel.com/vtune, Accessed January 2006.

[48] M. Itzkowitz, J. N. Wylie Brian, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *Proc. of the 2003 ACM/IEEE conference on Supercomputing*, pages 17–30, July 2003.

[49] K. Keutzer, S. Malik, A. R. Newton, M. Rabaey, and A Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer Aided Design for Integrated Circuits and Systems*, 19(12):1523–1542, December 2000.

[50] R. Lysecky, S. Cotterell, and F. Vahid. A fast on-chip profiler memory. In *Proc. of the 39th Conference on Design Automation*, pages 28–33, June 2002.

[51] R. Lysecky and F. Vahid. A study of the speedups and competitiveness of fpga soft processor cores using dynamic hardware/software partitioning. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, pages 18–23, March 2005.

[52] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, January 1998.

[53] F. Plavec, B. Fort, and Z. Vranesic. Experiences with soft-core processor design. In *Proc. of the 19th IEEE Conference on International Parallel and Distributed Processing Symposium*, pages 167b–167b, April 2005.

[54] P. Pop, P. Elese, and Z. Peng. *Analysis and Synthesis of Distributed Real-Time Embedded Systems.* Kluwer Academic Publishers, The Netherlands, 2004.

[55] M. Saini. Co-verification enhances time to market advantage of platform fpgas. Technical report, Mentor Graphics Corporation, July 2004.

[56] A. Sangiovanni-Vincentelli and G. Martin. Platform-based design and software design methodology for embedded systems. *Proc. of the IEEE on Design & Test of Computers*, 18:23–33, December 2001.

[57] F. Schirrmeister and A. Sangiovanni-Vincentelli. Virtual component co-design-applying function architecture co-design to automotive applications. In *Proc. of the 2001 Vehicle Electronics Conference*, pages 221–226, September 2001.

[58] B. Shah. Advanced call graph profiling techniques. Technical report, Intel Corporation, 2005.

[59] L. Shannon and P. Chow. Maximizing system performance: Using reconfigurability to monitor system communications. In *Proc. of the 2004 International Conference on Field Programmable Technology (ICFPT)*, pages 231–238, December 2004.

[60] L. Shannon and P. Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *Proc. of the 12th International Symposium on Field Programmable Gate Arrays*, pages 190–199, February 2004.

[61] Shannon, L. and Chow, P. Standardizing the Performance Assessment of Reconfigurable Processor Architectures. http://www.eecg.toronto.edu/ lesley/research/benchmarks/rates/shannon_rates.ps, Accessed May 2006.

[62] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, July-August 2002.

[63] G. Stitt, R. Lysecky, and F. Vahid. Dynamic hardware/software partitioning: A first approach. In *Proc. of the 40th Conference on Design Automation*, pages 250–255, June 2003.

[64] Sun Microsystems. Using UltraSPARC-IIICu Performance Counters to Improve Application Performance. http://developers.sun.com/prodtech/cc/articles/pcounters.html, Accessed February 2006.

[65] M. M. Tikir and J. K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *Proc. of the 2004 ACM/IEEE conference on Supercomputing*, pages 46–58, July 2003.

[66] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid. Soft-core processors for embedded systems. In *To Appear in Proc. of the 19th International Conference on Microelectronics*, December 2006.

[67] J. Turley. Embedded processors by the numbers. *Embedded Systems Programming*, May 1999.

[68] D. A Varley. Practical experience of the limitations of gprof. In *Software Practice and Experience*, pages 461–463, 1993.

[69] W. Wolf. *Principles of Embedded Computing System Design.* San Francisco, California, 2001.

[70] W. Wolf. A decade of hardware/software codesign. In *Proc. of the 5th International Symposium on Multimedia Software Engineering*, pages 38–43, December 2003.

[71] Xilinx Corporation. *Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Channel (FSL) Link*, May 2004.

[72] Xilinx Corporation. http://www.xilinx.com/, Accessed January 2005.

[73] Xilinx Corporation. *MicroBlaze Processor Reference Guide*, October 2005.

VITA AUCTORIS

Jason Gim Tong was born in Windsor, Ontario, Canada, on July 25, 1981. In 2000, he graduated from Vincent Massey Secondary School. From there after he attended the University of Windsor where he obtained his Bachelor of Applied Science (B.A.Sc) degree in Electrical and Computer Engineering (Computer Engineering Option) in 2004. He has sustained a position on the Dean's List throughout his undergraduate studies. He is currently a Master of Applied Science (M.A.Sc.) candidate in Electrical and Computer Engineering. His research interests include reconfigurable computing, hardware-software co-design for FPGA-based embedded systems, and digital system design. He was rewarded with a Tuition Waiver Scholarship (Fall 2005 to Summer 2006) from the University of Windsor. He is currently an IEEE student member.